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-2A、 SH2A-FPU

瑞萨32位RISC单片机

SuperH™ RISC engine系列

瑞萨电子

www.renesas.com

Rev.1.00 2008.12

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。

目　录

第 1 章 概要

1.1 SH-2A/SH2A-FPU 的特点

SH-2A/SH2A-FPU 是 32 位 RISC （精简指令系统计算机）单片机，其特点是与 SH-1、 SH-2、 SH-2E 单片机在目标代码级高位兼容。有无 FPU 的 SH-2A 和有 FPU 的 SH2A-FPU 两种。因为基本指令为 16 位长，可提高其编码效率、性能及使用的便利性。

SH-2A/SH2A-FPU 的特点如表 1.1 所示。

表 1.1 SH-2A/SH2A-FPU 的特点

项目	特点
CPU	• 瑞萨科技独创的体系结构 • 32 位内部数据总线 • 通用寄存器体系结构： — 16个32位通用寄存器 — 4个32位控制寄存器 — 4个32位系统寄存器 — 响应高速中断的寄存器存储体 • RISC 型指令系统 （与 SH 系列高位兼容）： — 指令长度：提高编码效率的16位基本指令与提高性能和使用便利性的32位指令 — 加载存储结构 — 延迟转移指令 — 以C语言为基础的指令系统 • 含 FPU 的 2 条指令同时执行型超标量 • 指令执行时间：最多 2 条指令 / 周期 • 地址空间：4G 字节 • 内置乘法器 • 5 段流水线 • 哈佛体系结构

项目	特点
浮点单元 （FPU）	• 内置浮点协处理器 • 支持单精度 （32 位）及双精度 （64 位） • 支持依照 IEEE754 的数据类型及异常 • 舍入模式：就近舍入及 0 方向舍入 • 非正规化数的处理：舍为 0 • 浮点寄存器 — 16个32位浮点寄存器 （单精度x16字或双精度x8字） — 2个32位浮点系统寄存器 • 支持 FMAC （乘法及累加）指令 • 支持 FDIV （除法运算） / FSQRT （平方根）指令 • 支持 FLDI0/FLDI1(加载常数 0/1) 指令 • 指令执行时间 — 等待时间(FMAC/FADD/FSUB/FMUL)：3个周期 （单精度）、 8个周期 （双精度） — 节距(FMAC/FADD/FSUB/FMUL)：1个周期 （单精度）、 6个周期 （双精度） 【注】 FMAC 只支持单精度。 • 5 段流水线

第 2 章 编程模型

2.1 数据格式

SH-2A/SH2A-FPU 所支持的数据格式如图 2.1 所示。

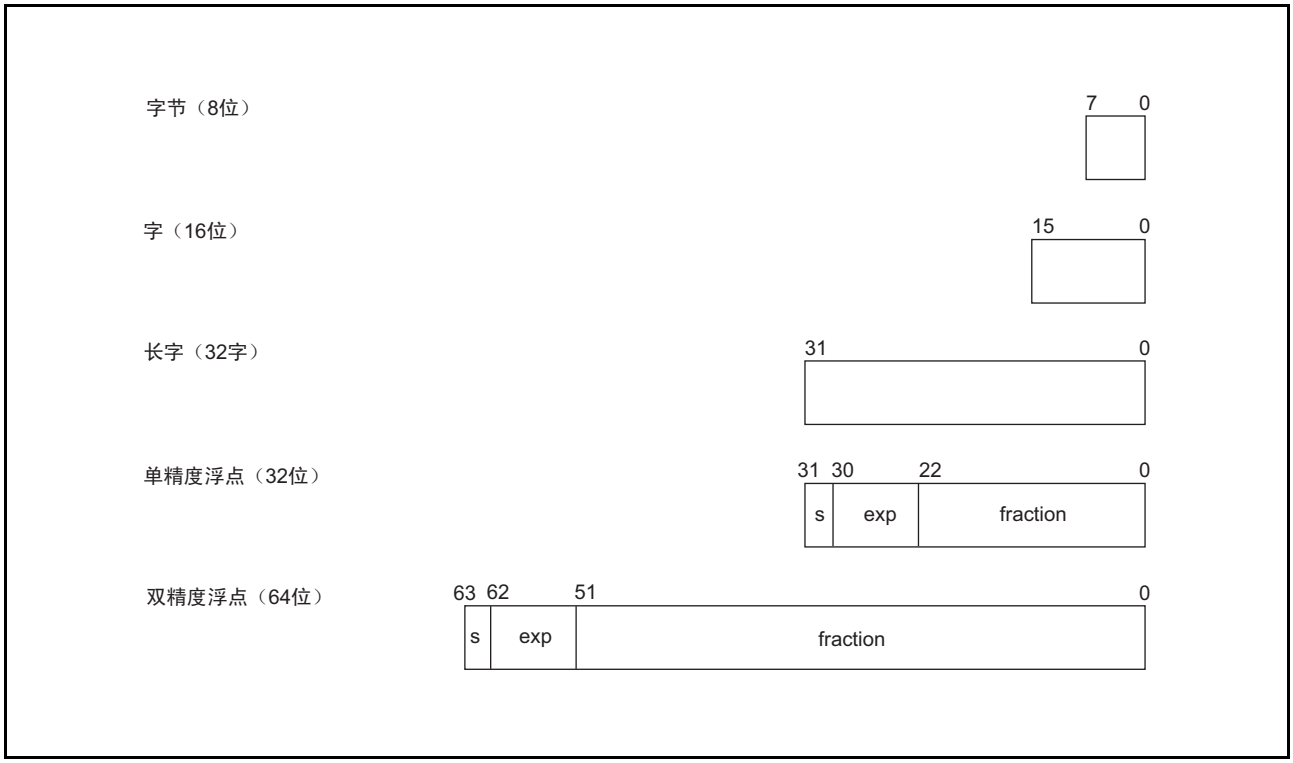


图 2.1 数据格式

2.2 寄存器的结构

2.2.1 通用寄存器

通用寄存器如图 2.2 所示。通用寄存器为 32 位长，从 R0 到 R15 共有 16 个，用于数据处理和地址计算。R0 也用作变址寄存器。有些指令能使用的寄存器被固定为 R0。 R15 用作硬件堆栈指针 （SP），通过使用 R15 参照堆栈，来执行异常处理时的状态寄存器 （SR）和程序计数器 （PC）的压栈或者出栈。

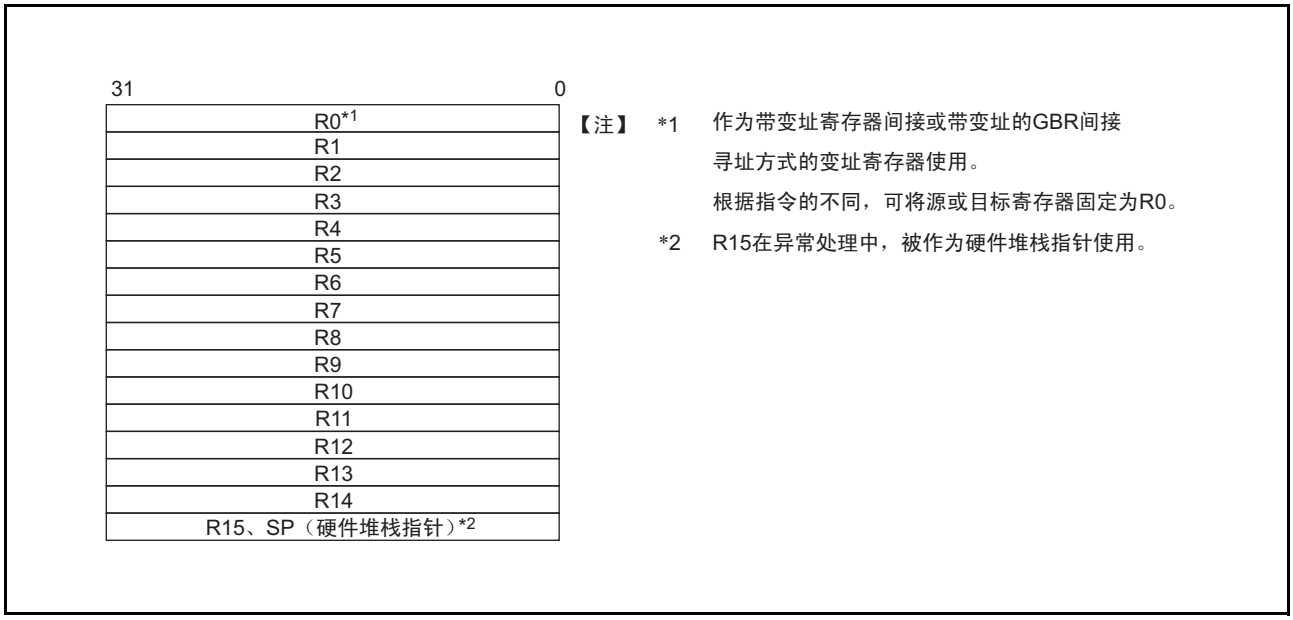


图 2.2 通用寄存器

2.2.2 控制寄存器

控制寄存器为 32 位长，有状态寄存器 （SR：Status register）、全局基址寄存器 （GBR：Global base register）、向量基址寄存器 （VBR：Vector base register）、跳转表基址寄存器 （TBR：Jump table base register）共四个。

SR 寄存器表示各指令的处理状态。

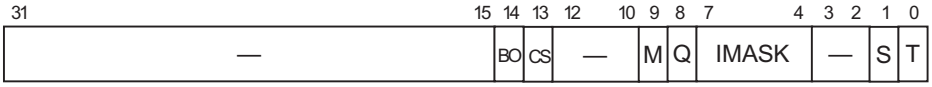
GBR 寄存器用作 GBR 间接寻址方式的基址，并用于内部外围模块寄存器的数据传送等。

VBR 寄存器用作包括中断的异常处理向量区域的基址。

TBR 寄存器用作函数表区域的基址。

(1) 状态寄存器 SR

（32 位、初始值 =0000 0000 0000 0000 00X0 00XX 1111 00XX(X ＝不定)）



【注】—：保留位。读取时只读0，写入值也只为0。

- BO：表示寄存器存储体上溢。
- CS：表示由于执行 CLIP 指令，超过饱和上限值或低于饱和下限值。
- M、 Q：DIV0S、 DIV0U、 DIV1 指令下使用。
- IMASK：中断屏蔽级
- S：指定 MAC 指令的饱和运行。
- T：真 / 假条件或进 / 借位

(2) 全局基址寄存器 GBR （32 位、初始值 = 不定）

GBR 作为 GBR 参照 MOV 指令的基址被参照。

(3) 向量基址寄存器 VBR （32 位、初始值 =H'0000 0000）

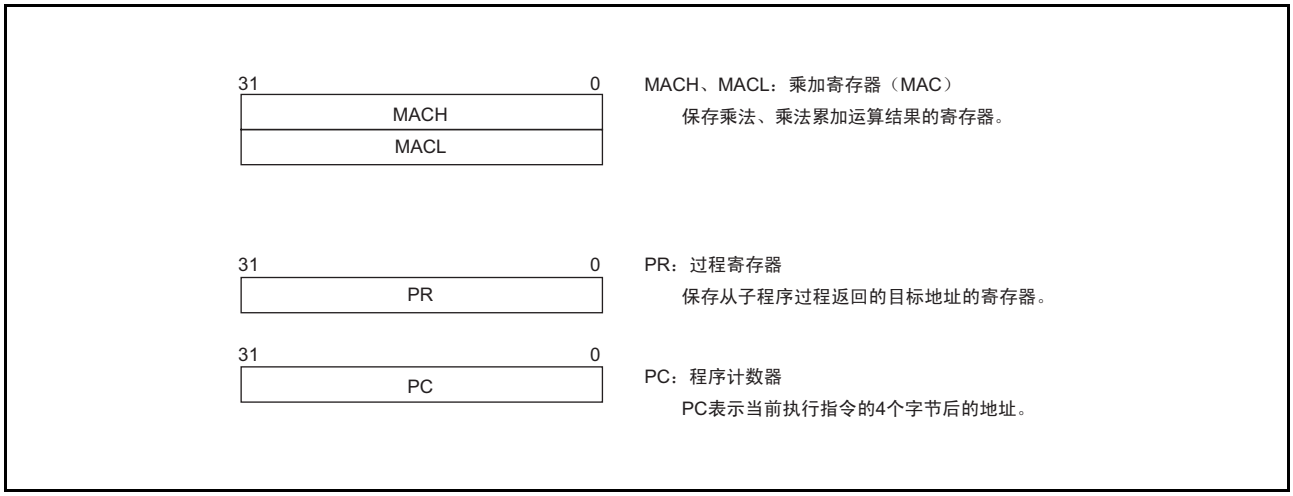
VBR 在产生异常及中断时，作为转移目标的基址被参照。

(4) 跳转表基址寄存器 TBR （32 位、初始值 = 不定）

在表参照子程序调用指令 JSR/N @@(disp8,TBR) 时，TBR 作为配置给存储器的函数表的起始地址被参照。

2.2.3 系统寄存器

系统寄存器为 32 位长，有乘加寄存器 （MACH、 MACL）、过程寄存器 （PR）和程序计数器 （PC）共 4 个。MACH、MACL 保存乘法或者乘法累加运算的结果。PR 保存从子程序过程返回的目标地址。PC 表示执行中的程序地址，控制处理的流程。



(1) 乘加高位寄存器 MACH（32 位、初始值 = 不定）、
乘加低位寄存器 MACL （32 位、初始值 = 不定）

MACH/MACL 可作为 MAC 指令的加法运算值使用。另外也用于保存 MAC 指令、MUL 指令的运算结果。

(2) 过程寄存器 PR （32 位、初始值 = 不定）

PR 保存使用 BSR、BSRF、JSR 指令的子程序调用的返回地址。PR 通过子程序的返回指令 (RTS) 被参照。

(3) 程序计数器 PC （32 位、初始值为向量表中的 PC 值）

PC 表示执行中的指令地址。

2.2.4 浮点寄存器

浮点寄存器如图 2.3 所示。有 32 位的浮点寄存器 FPR0 ～ FPR15 共 16 个。这 16 个寄存器作为 FR0 ～ FR15、 DR0/2/4/6/8/l0/12/14 被参照。 FPRn 与参照名的对照由 FPSCR 的 PR 位和 SZ 位决定。请参照图 2.3。

(1) 浮点寄存器 FPRn （16 个 寄存器）

FPR0、 FPR l、 FPR2、 FPR3、 FPR4、 FPR5、 FPR6、 FPR7、
FPR8、 FPR9、 FPR10、 FPR11、 FPR12、 FPR13、 FPR14、 FPR15

(2) 单精度浮点寄存器 FRi （16 个寄存器）

FR0 ～ FR15 分配在 FPR0 ～ FPR15。

(3) 双精度浮点寄存器或单精度浮点寄存器对 DRi （8 个寄存器）

DR 寄存器由 2 个 FR 寄存器组成。
DR0 = {FPR0, FPR1}、 DR2 = {FPR2, FPR3 }、
DR4 = {FPR4, FPR5}、 DR6 = {FPR6, FPR7}、
DR8 = {FPR8, FPR9}、 DR10 = {FPR10, FPR11}、
DR12 = {FPR12, FPR13}、 DR14 = {FPR14, FPR15}

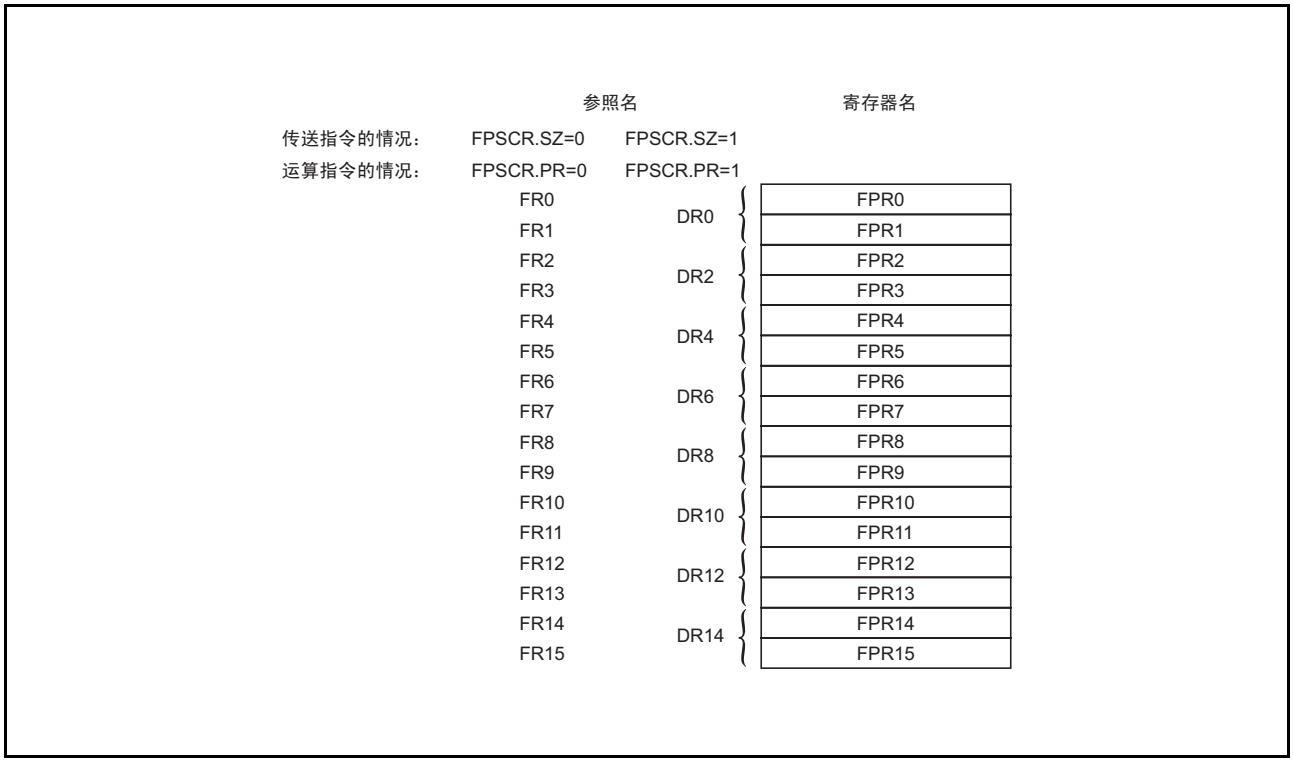


图 2.3 浮点寄存器

【编程时的注意】
复位后 FPR0 ～ FPR15 的值不定。

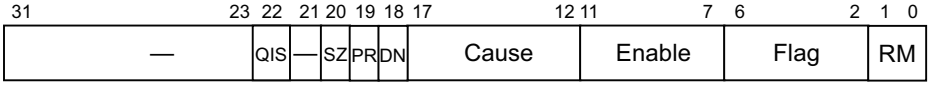
2.2.5 浮点系统寄存器

(1) 浮点通信寄存器 FPUL （32 位、初始值 = 不定）

通过 FPUL 进行 FPU 寄存器与 CPU 寄存器间的数据传送。

(2) 浮点状态 / 控制寄存器 FPSCR

（32 位、初始值 =H'0004 0001）



QIS：将 qNaN 或者 ± ∞ 作为 sNaN 处理。仅在 FPSCR 允许 V=1 时有效。

QIS=0：作为 qNaN 或者 ± ∞ 处理。

QIS=1：产生异常 （与 sNaN 的处理相同）。

SZ：传送长度模式

SZ=0：FMOV 指令的数据长度为 32 位。

SZ=1：FMOV 指令的数据长度为 32 位对 （64 位）。

PR：精度模式

PR=0：浮点指令以单精度执行。

PR=1：浮点指令以双精度执行 （不支持双精度的指令的结果为未定义。）

DN：非正规化模式 （总是为 1）

DN=1：非正规化数当作 0 处理。

Cause：FPU 异常源字段

Enable：FPU 异常允许字段

Flag：FPU 异常标志字段

		FPU 错误 (E)	无效运算 (V)	被 0 除 (Z)	上溢 (O)	下溢 (U)	不正确 (I)
Cause	FPU 异常源字段	bit17	bit16	bit15	bit14	bit13	bit12
Enable	FPU 异常允许字段	—	bit11	bit10	bit9	bit8	bit7
Flag	FPU 异常标志字段	—	bit6	bit5	bit4	bit3	位 2

执行 FPU 运算指令时， FPU 异常源字段最初被设定为 0。之后发生 FPU 异常时， FPU 异常源字段与 FPU 异常标志字段的相应位置 1。

FPU 异常标志字段保持 FPU 异常标志字段最后被清除之后发生的异常状态。

RM： 舍入模式

RM=00：就近舍入

RM=01：向0方向舍入

RM=10：保留

RM=11：保留

bit21、 23 ～ 31：保留

【注】 SH-2A 不会发生 FPU 错误。

2.2.6 寄存器存储体

通用寄存器 R0 ～ R14、控制寄存器 GBR、系统寄存器 MACH、 MACL、 PR 共 19 个 32 位寄存器可以使用寄存器存储体进行高速寄存器保存或者返回。对储存体的保存在 CPU 接受使用寄存器存储体的中断后自动执行。通过中断处理程序发行 RESBANK 指令进行来执行从存储体的返回。

详细内容请参照 “第 7 章 寄存器存储体”。

2.2.7 寄存器的初始值

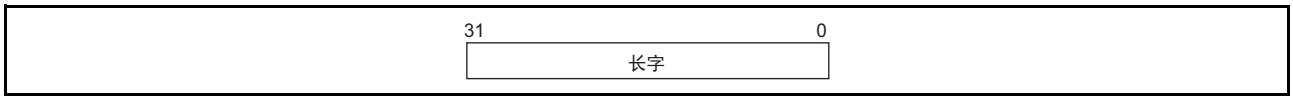
表 2.1 寄存器的初始值

区分	寄存器	初始值
通用寄存器	R0 ～ R14	不定
	R15(SP)	向量地址表中的 SP 值。
控制寄存器	SR	I3 ～ I0 为 1111(H'F)， BO、 CS 为 0，保留位为 0，其它不定。
	GBR、 TBR	不定
	VBR	H'00000000
系统寄存器	MACH、 MACL、 PR	不定
	PC	向量地址表中的 PC 值。
浮点寄存器	FPR0 ～ FPR15	不定
浮点系统寄存器	FPUL	不定
	FPSCR	H'00040001

2.3 数据格式

2.3.1 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）或者字 （16 位）时，符号扩展为长字后，保存到寄存器。



2.3.2 存储器的数据格式

数据格式有字节、字和长字。存储器能够以 8 位字节、 16 位字、 32 位长字中的任意格式存取。不足 32 位的存储器操作数在符号扩展或者零扩展后，保存到寄存器。

字操作数从字边界 （每 2 个字节的偶数地址：地址 2n）存取，长字操作数从长字边界 （每 4 个字节的偶数地址：地址 4n）存取。不符合此情况的为地址错误。字节操作数可从任意地址进行存取。

数据格式只能选择大端法的字节顺序。

存储器的数据格式如图 2.4 所示。

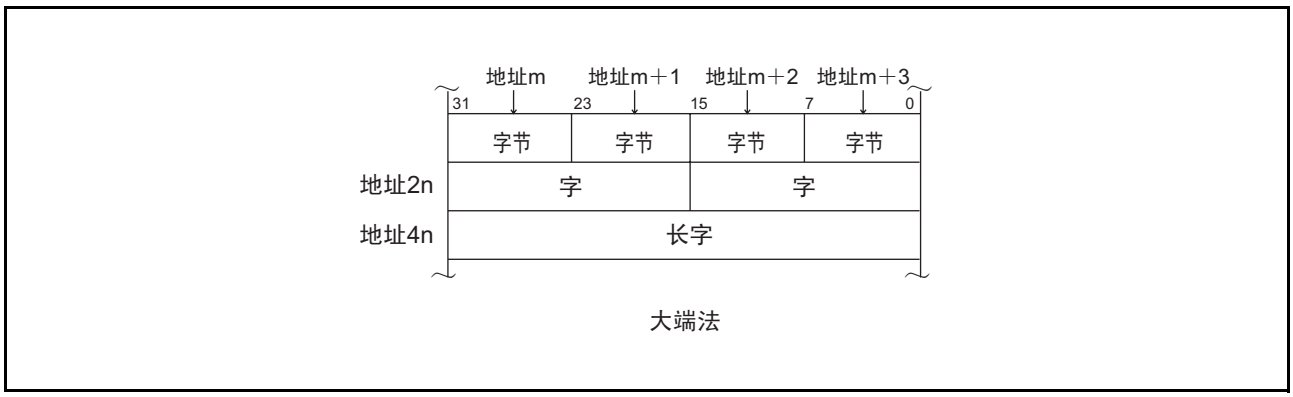


图 2.4 存储器的数据格式

2.3.3 立即数的数据格式

字节的立即数分配在操作码中。

MOV、 ADD、 CMP/EQ 指令时将立即数进行符号扩展后通过寄存器和长字进行运算。而另一方面 TST、 AND、 OR、 XOR 指令时将立即数进行零扩展后以长字进行运算。因此，如果 AND 指令时使用立即数，目标寄存器的高 24 位总是被清除。

20 位的立即数分配在 32 位长的传送指令 MOVI20 及 MOVI20S 的代码中。 MOVI20 指令将立即数符号扩展并保存在目标寄存器。 MOVI20S 指令将立即数向高位移 8 位，进行符号扩展并保存在目标寄存器。

字和长字的立即数不分配在操作码中，而分配在存储器表中。通过带位移量的 PC 相对寻址方式的立即数传送指令 （MOV），参照存储器表。

有关具体例子，请参照 “**第 4 章 指令的特点**”的 “**4.1(10) 立即数**”。

2.4 处理状态

CPU 的处理状态有复位状态、异常处理状态、总线权释放状态、程序执行状态、低功耗状态 5 种。状态间的转移如**图 2.5** 所示。

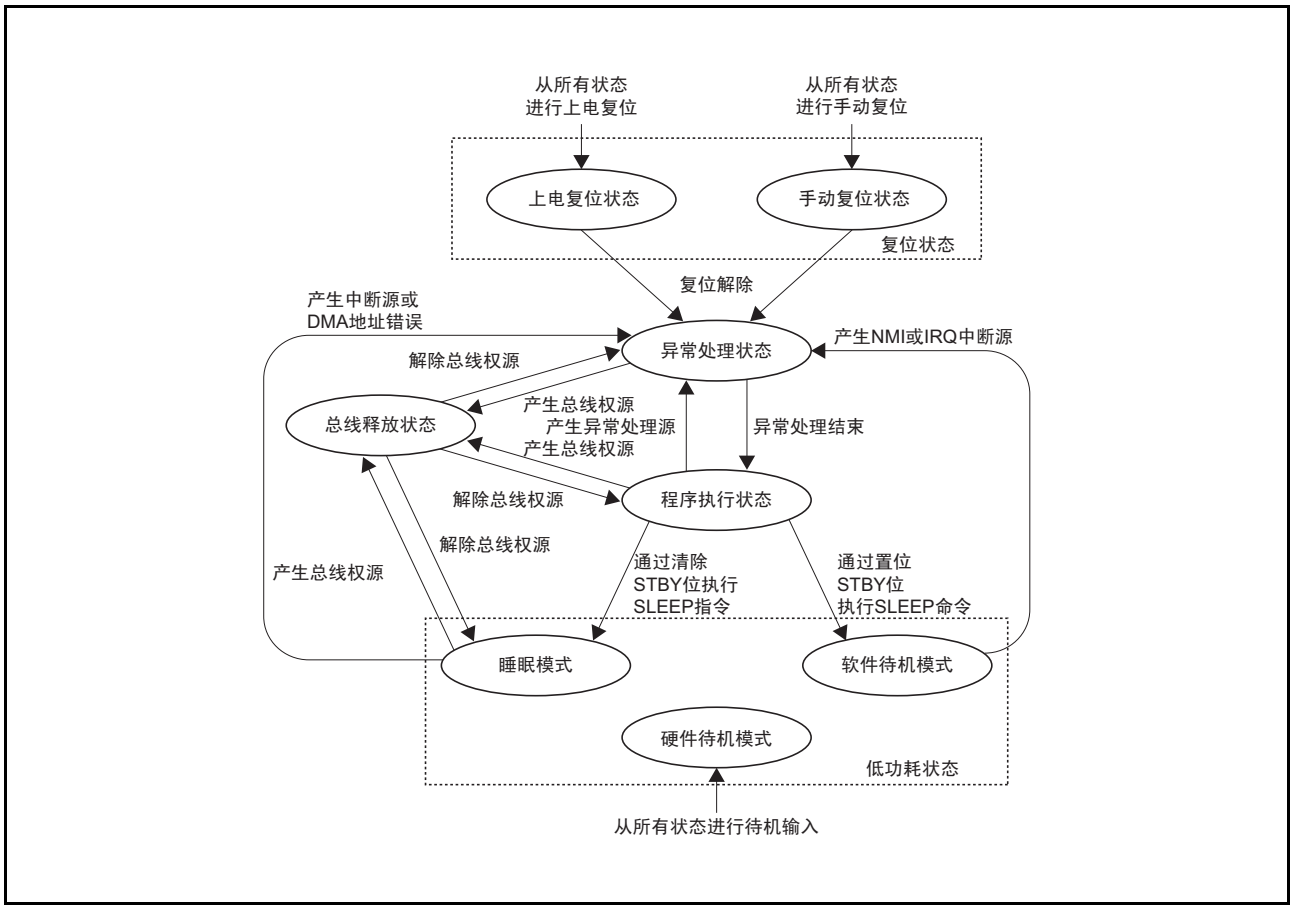


图 2.5 处理状态的状态转移图

(1) 复位状态

为 CPU 被复位的状态。复位有上电复位和手动复位两种。详细内容请参照硬件手册。

(2) 异常处理状态

根据复位和中断等异常处理源， CPU 为改变处理状态流程时的过渡状态。

复位时，从异常处理向量表取出作为程序计数器 （PC）初始值的执行起始地址与堆栈指针 （SP）的初始值并分别保存，转移到起始地址并开始执行程序。

中断时，参照 SP，将 PC 与状态寄存器 （SR）保存到堆栈区。从异常处理向量表取出异常服务程序的起始地址，转移到该地址并开始执行程序。

之后，处理状态变为程序执行状态。

(3) 程序执行状态

CPU 依次执行程序的状态。

(4) 低功耗状态

CPU 停止运行功耗低的状态。通过睡眠指令变为睡眠模式或软件待机模式。输入硬件待机输入时变为硬件待机模式。

(5) 总线权释放状态

CPU 对请求总线权的器件释放总线的状态。

【注】 处理状态详情请参照各产品的硬件说明书。

第 3 章 异常处理

3.1 概要

3.1.1 异常处理的种类和优先顺序

如表 3.1 所示，由于复位、地址错误、 RAM 错误、寄存器存储体错误、中断和指令的各异常源启动异常处理。如表 3.1 所示，异常源设有优先顺序，如果同时产生多个异常源，就按照此优先顺序接受并处理。

表 3.1 异常源的种类和优先顺序

	异常处理	优先顺序
复位	上电复位	高
	手动复位	
地址错误	CPU 地址错误	
	DMAC 地址错误	
RAM 错误	RAM 错误	
指令	FPU 异常	
	整数除法异常 （被 0 除）	
	整数除法异常 （上溢）	
寄存器存储体错误	存储体下溢	
	存储体上溢	
中断	NMI	
	用户断点	
	H-UDI	
	外部中断 (IRQ)	
	内置外围模块	
指令	陷阱指令 （TRAPA 指令）	
	一般非法指令 （未定义代码）	
	槽非法指令 （配置在紧接着延迟转移指令 *1 之后的未定义代码 （包括与 FPU 模块待机或无 FPU 产品的 FPU 指令及 FPU 相关的 CPU 指令，无寄存器存储体产品的寄存器存储体相关指令 *2）、改写 PC 指令 *3、 32 位指令 *4、 RESBANK 指令、 DIVS 指令或者 DIVU 指令）	低

【注】 *1 延迟转移指令：JMP、 JSR、 BRA、 BSR、 RTS、 RTE、 BF/S、 BT/S、 BSRF、 BRAF

*2 寄存器存储体相关指令：RESBANK、 LDBANK、 STBANK

*3 改写 PC 指令：JMP、JSR、BRA、BSR、RTS、RTE、BT、BF、TRAPA、BF/S、BT/S、BSRF、BRAF、JSR/N、 RTV/N

*4 32 位指令：BAND.B、 BANDNOT.B、 BCLR.B、 BLD.B、 BLDNOT.B、 BOR.B、 BORNOT.B、 BSET.B、BST.B、 BXOR.B、 FMOV.S@disp12、 FMOV.D@disp12、 MOV.B@disp12、 MOV.W@disp12、MOV.L@disp12、 MOVI20、 MOVI20S、 MOVU.B、 MOVU.W

3.1.2 异常处理的运行

在如表 3.2 所示的时序中检测到各异常源后开始处理。

表 3.2 异常源检测和异常处理的开始时序

异常处理		异常源检测和异常处理的开始时序
复位	上电复位	在检测出上电复位条件时开始。
	手动复位	在检测出手动复位条件时开始。
地址错误		在指令的解码时检测，并在执行中的指令结束后开始处理。
RAM 错误		
中断		
寄存器存储体错误	存储体下溢	在没有向寄存器存储体进行保存时，执行 RESBANK 指令时开始处理。
	存储体上溢	将中断控制器设定为接受寄存器存储体上溢异常，产生使用寄存器存储体的中断且被 CPU 接受，以及向寄存器存储体的全部区域进行保存后开始处理。
指令	陷阱指令	通过执行 TRAPA 指令开始处理。
	一般非法指令	当延迟转移指令 （延迟槽）以外的未定义代码 （包括与 FPU 模块待机或无 FPU 产品的 FPU 指令及 FPU 相关的 CPU 指令，无寄存器存储体产品的寄存器存储体相关指令）被解码时开始处理。
	槽非法指令	配置在紧接着延迟转移指令之后 （延迟槽）的未定义代码 （包括与 FPU 模块待机或无 FPU 产品的 FPU 指令及 FPU 相关的 CPU 指令，无寄存器存储体产品的寄存器存储体相关指令）、改写 PC 指令、32 位指令、RESBANK 指令、DIVS 指令或者 DIVU 指令被解码时开始处理。
	整数除法指令	检测出被零除的异常或者由于用 –1 除负数的最大值 （H'80000000）产生上溢异常时，开始处理。
	浮点运算指令	由于浮点运算指令的无效运算异常 （IEEE754 规则）、被零除的异常、上溢、下溢或不正确异常开始处理。另外， FPSCR 的 QIS 位被置位时，如果将 qNaN 或 ± ∞ 输入到浮点运算指令源时就开始处理。

启动异常处理时， CPU 的运行如下：

(1) 由复位引起的异常处理

从异常处理向量表中 （PC 和 SP 在上电复位时分别从地址 H'00000000 和 H'00000004 ；手动复位时分别从地址 H'00000008 和 H'0000000C）取出程序计数器 （PC）和堆栈指针 （SP）的初始值。有关异常处理向量表，请参照 “3.1.3 异常处理向量表”。然后，将向量基址寄存器 （VBR）初始化为 H'00000000，将状态寄存器 （SR）的中断屏蔽位 （I3 ～ I0）初始化为 H'F （1111），将 BO 位及 CS 位初始化为 0。另外将 INTC 的 IBNR 的 BN 位初始化为 0。并且在上电复位时，将具有 FPU 产品的 FPSCR 初始化为 H'00040001。从异常处理向量表中取出的 PC 地址开始执行程序。

(2) 由地址错误、 RAM 错误、寄存器存储体错误、中断和指令引起的异常处理

将 SR 和 PC 保存到 R15 指示的堆栈。 NMI 以及 UBC 以外的中断异常处理中，如果进行了使用寄存器存储体的设定，便将通用寄存器 R0 ～ R14、控制寄存器 GBR、系统寄存器 MACH、 MACL、 PR 以及被执行的中断异常处理的向量表地址偏移量保存到寄存器存储体。如果是由于地址错误、 RAM 错误、寄存器存储体错误、 NMI 中断、 UBC 中断、指令引起的异常处理，就不进行向寄存器存储体的保存。另外，如果已经向寄存器存储体的所有存储体保存，就自动保存到堆栈而不是寄存器存储体。此时，需要在中断控制器中设定不接受寄存器存储体上溢异常。如果设定接受寄存器存储体上溢异常，就会发生寄存器存储体上溢异常。在中断异常处理时，将中断优先级写到 SR 的 I3 ～ I0。由地址错误、 RAM 错误、指令引起的异常处理时， I3 ～ I0 位不受影响。然后，从异常处理向量表中取出起始地址，并从该地址开始执行程序。

3.1.3 异常处理向量表

执行异常处理前，需要预先将异常处理向量表设定到存储器，并将异常服务程序的起始地址保存到异常处理向量表中 （将 PC 和 SP 的初始值保存到复位异常处理表中）。

分别给各异常源分配了不同的向量号和向量表地址偏移量，从对应的向量号和向量表地址偏移量算出向量表地址。在异常处理中，从此向量表地址指示的异常处理向量表中取出异常服务程序的起始地址。

向量号和向量表地址偏移量如**表** 3.3 所示，向量表地址的计算方法如**表** 3.3 所示。

表 3.3 异常处理向量表

异常源		向量号	向量表地址偏移量
上电复位	PC	0	H'00000000 ～ H'00000003
	SP	1	H'00000004 ～ H'00000007
手动复位	PC	2	H'00000008 ～ H'0000000B
	SP	3	H'0000000C ～ H'0000000F
一般非法指令		4	H'00000010 ～ H'00000013
RAM 错误		5	H'00000014 ～ H'00000017
槽非法指令		6	H'00000018 ～ H'0000001B
（系统保留）		7	H'0000001C ～ H'0000001F
		8	H'00000020 ～ H'00000023
CPU 地址错误		9	H'00000024 ～ H'00000027
DMAC 地址错误		10	H'00000028 ～ H'0000002B
中断	NMI	11	H'0000002C ～ H'0000002F
	用户断点	12	H'00000030 ～ H'00000033
FPU 异常		13	H'00000034 ～ H'00000037
H-UDI		14	H'00000038 ～ H'0000003B
存储体上溢		15	H'0000003C ～ H'0000003F
存储体下溢		16	H'00000040 ～ H'00000043
整数除法异常 （被 0 除）		17	H'00000044 ～ H'00000047
整数除法异常 （上溢）		18	H'00000048 ～ H'0000004B
（系统保留）		19 • 31	H'0000004C ～ H'0000004F • H'0000007C ～ H'0000007F
陷阱指令 （用户向量）		32 • 63	H'00000080 ～ H'00000083 • H'000000FC ～ H'000000FF
外部中断 （IRQ）、内部外围模块 *		64 • 511	H'00000100 ～ H'00000103 • H'000007FC ～ H'000007FF

【注】 * 有关外部中断、各内部外围模块中断的向量号以及向量表偏移量，请参照各产品的硬件手册的 “中断控制器” 中的 “中断异常处理向量和优先顺序”。

表 3.4 异常处理向量表地址的计算方法

异常源	向量表地址的计算方法
复位	向量表地址 = （向量表地址偏移量） = （向量号） × 4
地址错误、 RAM 错误、 寄存器存储体错误，中断、指令	向量表地址 =VBR + （向量表地址偏移量） =VBR + （向量号） × 4

【注】 VBR：向量基址寄存器
向量表地址偏移量：参照表3.3
向量号：参照表3.3

3.2 复位

3.2.1 复位的种类

复位是优先顺序最高的异常处理源，有上电复位和手动复位 2 种。在上电复位或者手动复位中， CPU 状态都被初始化。FPU 状态在上电复位时被初始化，而在手动复位时不被初始化。关于内部外围模块、PFC、IO 端口的状态，请参照各产品的硬件手册。

3.2.2 上电复位

检测出上电复位条件时，进入上电复位状态。关于上电复位条件，请参照各产品硬件手册的 “异常处理”中的 “上电复位”。

解除上电复位状态时，就开始上电复位的异常处理。此时 CPU 的运行如下：

(1) 从异常处理向量表中取出程序计数器 （PC）的初始值 （执行起始地址）。
(2) 从异常处理向量表中取出堆栈指针 （SP）的初始值。
(3) 将向量基址寄存器 （VBR）清除为H'00000000，将状态寄存器 （SR）的中断屏蔽位 （I3～I0）初始化为H'F （1111），将BO位以及CS位初始化为0。另外，将INTC的IBNR的BN位初始化为 0。并且，将具有FPU产品的FPSCR初始化为H'00040001。
(4) 从异常处理向量表中取出的值分别设定到PC和SP，然后开始执行程序。

另外，必须在接通系统电源时进行上电复位处理。

3.2.3 手动复位

检测出手动复位条件时，进入手动复位状态。关于手动复位条件，请参照各产品的硬件手册的 “异常处理中的 “手动复位”。

手动复位开始时 CPU 的运行如下：

(1) 从异常处理向量表中取出程序计数器 （PC）的初始值 （执行起始地址）。
(2) 从异常处理向量表中取出堆栈指针 （SP）的初始值。
(3) 将向量基址寄存器 （VBR）清除为H'00000000，将状态寄存器 （SR）的中断屏蔽位 （I3～I0）初始化为H'F （1111），将BO位以及CS位初始化为0。另外，将INTC的IBNR的BN位初始化为 0。
(4) 从异常处理向量表中取出的值分别设定到PC和SP，然后开始执行程序。

发生手动复位时，保持总线周期。在总线权释放中或者 DMAC 突发传送中发生手动复位时，到 CPU 获得总线权之前保留手动复位异常处理。但是如果从发生手动复位到总线周期结束的这段时间，超过内部手动复位时间的一定周期，就不保留内部手动复位源并忽略，因此不发生手动复位异常处理。详细内容，请参照各产品硬件手册的 “异常处理”中的 “手动复位”。

手动复位中初始化 CPU 及 INTC 的 IBNR 的 BN 位。不初始化 FPU 和其他模块。

3.3 地址错误

3.3.1 地址错误发生源

如表 3.5 所示，在取指令或者读写数据时发生地址错误。

表 3.5 总线周期和地址错误

总线周期		总线周期的内容	地址错误的发生
种类	总线主控器		
取指令	CPU	从偶数地址取指令	无 （正常）
		从奇数地址取指令	发生地址错误
		从非内部外围模块空间 * 取指令	无 （正常）
		从内部外围模块空间 * 取指令	发生地址错误
		单芯片模式时，从外部存储器空间取指令	发生地址错误
读写数据	CPU 或者 DMAC	从偶数地址存取字数据	无 （正常）
		从奇数地址存取字数据	发生地址错误
		从长字边界存取长字数据	无 （正常）
		从双长字边界存取双长字数据	无 （正常）
		从非双长字边界存取双长字数据	发生地址错误
		在 16 位内部外围模块空间 * 内存取长字数据	无 （正常）
		在 8 位内部外围模块空间 * 内存取长字数据	无 （正常）
		单芯片模式时，存取外部存储空间	发生地址错误

【注】 * 关于内部外围模块空间，请参照各产品硬件手册的 “总线状态控制器”。

3.3.2 地址错误异常处理

当发生地址错误时，引起地址错误的总线周期就结束，并在执行中的指令结束后开始地址错误的异常处理。此时 CPU 的运行如下：

(1) 从异常处理向量表中取出对应发生地址错误的异常服务程序的起始地址。
(2) 将状态寄存器 （SR）保存到堆栈。
(3) 将程序计数器 （PC）保存到堆栈。保存的PC值是最后已执行指令的下一条指令的起始地址。
(4) 转移至从异常处理向量表中取出的地址，开始执行程序。此时的转移不是延迟转移。

3.4 RAM 错误

3.4.1 RAM 错误发生源

内部 RAM 的读操作时，如果发生软错误就发生 RAM 错误。详细内容，请参照各产品硬件手册的 “异常处理”中的 “RAM 错误”。

3.4.2 RAM 错误异常处理

发生 RAM 错误时，引起 RAM 错误的总线周期就结束，并在执行中的指令结束后开始 RAM 错误异常处理。此时 CPU 的运行如下：

(1) 从异常处理向量表中取出对应发生RAM错误的异常服务程序的起始地址。
(2) 将状态寄存器 （SR）保存到堆栈。
(3) 将程序计数器 （PC）保存到堆栈。保存的PC值是最后已执行指令的下一条指令的起始地址。
(4) 转移至从异常处理向量表中取出的地址转移，开始执行程序。此时的转移不是延迟转移。

3.5 寄存器存储体错误

3.5.1 寄存器存储体错误发生源

(1) 存储体上溢
在中断控制器设定为接受寄存器存储体上溢异常，发生使用寄存器存储体的中断且被CPU接受时，已经执行向寄存器存储体所有区域保存的情况。
(2) 存储体下溢
在没有执行向寄存器存储体保存时，执行RESBANK指令时的情况。

3.5.2 寄存器存储体错误异常处理

发生寄存器存储体错误时，开始寄存器存储体错误异常处理。此时 CPU 的运行如下：

(1) 从异常处理向量表中取出对应发生寄存器存储体错误的异常服务程序的起始地址。
(2) 将状态寄存器 （SR）保存到堆栈。
(3) 将程序计数器 （PC）保存到堆栈。保存的PC值在存储体上溢时是最后已执行指令的下一条指令的起始地址，在存储体下溢时是已执行的RESBANK指令的起始地址。为防止存储体上溢时的多重中断，将成为存储体上溢源的中断级别写入状态寄存器 （SR）的中断屏蔽级别位 （I3～I0）。
(4) 转移至从异常处理向量表中取出的地址，开始执行程序。此时的转移不是延迟转移。

3.6 中断

3.6.1 中断源

如表 3.6 所示，启动中断异常处理的中断源有 NMI、用户断点、 H-UDI、外部中断和内部外围模块。

表 3.6 中断源

种类	请求源	源数
NMI	NMI 引脚 （来自外部的输入）	1
用户断点	用户断点控制器	1
H-UDI	用户调试接口	1
外部中断 (IRQ)、内部外围模块	外部中断引脚、内部外围模块 *	*

分别给各中断源分配了不同的向量号和向量表偏移量。有关向量号和向量表地址偏移量，请参照各产品硬件手册的 “中断控制器”中的 “中断异常向量和优先顺序”。

【注】 * 关于外部中断 （IRQ）、内部外围模块的请求源和中断源数，请参照各产品硬件手册的 “中断控制器”中的 “中断源”。

3.6.2 中断优先顺序

中断源设有优先顺序，如果同时发生多个中断 （多重中断），就通过中断控制器 （INTC）判断优先顺序，并按照该判断结果启动异常处理。

用优先级 0 ～ 16 表示中断源的优先顺序，优先级 0 为最低、优先级 16 为最高。 NMI 中断是优先级为 16 并且不能屏蔽的最高级中断，总是被接受。用户断点中断以及 H-UDI 的优先级为 15。能通过 INTC 的中断优先级设定寄存器自由设定 IRQ 中断和内部外围模块中断的优先级 （表 3.7）。能设定的优先级为 0 ～ 15，不能设定优先级 16。关于中断优先级设定寄存器，请参照各产品硬件手册的 “中断控制器”。

表 3.7 中断优先顺序

种类	优先级	备 注
NMI	16	固定优先级、不能屏蔽
用户断点	15	固定优先级
H-UDI	15	固定优先级
外部中断 (IRQ)、内部外围模块	0 ～ 15	通过中断优先级设定寄存器进行设定

3.6.3 中断异常处理

如果产生中断，就通过中断控制器 （INTC）判断优先顺序。 NMI 总是被接受，但是其他中断只在其优先级高于设定在状态寄存器 （SR）的中断屏蔽位 （I3 ～ I0）的优先级时才被接受。

如果接受中断，就开始中断异常处理。中断异常处理中， CPU 将 SR 和程序计数器 （PC）保存到堆栈。NMI 及 UBC 以外的中断异常处理中，如果进行了使用寄存器存储体的设定，通用寄存器 R0 ～ R14、控制寄存器 GBR、系统寄存器 MACH、 MACL、 PR 以及被执行的异常处理的向量表地址偏移量保存到寄存器存储体。如果是由于地址错误、 NMI 中断、 UBC 中断指令引起的异常处理，就不向寄存器存储体保存。同时，如果已执行向寄存器存储体的所有存储体保存，就自动保存到堆栈而不是寄存器存储体。此时，需要在中断控制器中设定不接受寄存器存储体上溢异常。如果设定接受寄存器存储体上溢异常，就会发生寄存器存储体上溢异常。然后将接受的中断优先级的值写到 SR 的 I3 ～ I0 位。但是 NMI 优先级为 16，而设定到 I3 ～ I0 位的值为 H'F （级 15）。接着，从对应被接受中断的异常处理向量表中取出异常服务程序的起始地址，转移到该地址后开始执行程序。有关中断异常处理的详细内容请参照各产品硬件手册的 “中断控制器”中的 “运行说明”。

3.7 由指令引起的异常

3.7.1 由指令引起的异常的种类

如表 3.8 所示，启动异常处理的指令有陷阱指令，槽非法指令，一般非法指令，整数除法指令，浮点运算指令。

表 3.8 由指令引起的异常种类

种类	源指令	备 注
陷阱指令	TRAPA	
槽非法指令	配置在紧接着延迟转移指令 （延迟槽）之后的未定义代码 （包括与 FPU 模块待机或无 FPU 产品的 FPU 指令及 FPU 相关的 CPU 指令，无寄存器存储体产品的寄存器存储体相关指令）、改写 PC 指令、 32 位指令、 RESBANK 指令、 DIVS 指令或者 DIVU 指令	延迟转移指令：JMP、JSR、BRA、BSR、RTS、RTE、BF/S、 BT/S、 BSRF、 BRAF 寄存器存储体相关指令：RESBANK、 LDBANK、 STBANK 改写 PC 指令：JMP、JSR、BRA、BSR、RTS、RTE、BT、BF、TRAPA、BF/S、BT/S、BSRF、BRAF、JSR/N、RTV/N 32 位指令：BAND.B、 BANDNOT.B、 BCLR.B、 BLD.B、BLDNOT.B、 BOR.B、 BORNOT.B、 BSET.B、 BST.B、BXOR.B、 FMOV.S@disp12、 FMOV.D@disp12、MOV.B@disp12、 MOV.W@disp12、 MOV.L@disp12、MOVI20、 MOVI20S、 MOVU.B、 MOVU.W
一般非法指令	延迟槽以外的未定义代码 （包括与 FPU 模块待机或无 FPU 产品的 FPU 指令及 FPU 相关的 CPU 指令，无寄存器存储体产品的寄存器存储体相关指令）。	
整数除法异常	被零除	DIVU、 DIVS
	负的最大值 ÷(–1)	DIVS
浮点运算指令	引起 IEEE754 规格定义的无效运算异常或者被零除异常的指令，以及可能引起上溢、下溢或不正确异常的指令	FADD、FSUB、FMUL、FDIV、FMAC、FCMP/EQ、FCMP/GT、 FLOAT、 FTRC、 FCNVDS、 FCNVSD、 FSQRT

3.7.2 陷阱指令

如果执行 TRAPA 指令，就开始陷阱指令的异常处理。此时 CPU 的运行如下：

(1) 从异常处理向量表中取出TRAPA指令指定的向量号对应的异常服务程序的起始地址。
(2) 将状态寄存器 （SR）保存到堆栈。
(3) 将程序计数器 （PC）保存到堆栈。保存的PC值是TRAPA指令的下一条指令的起始地址。
(4) 转移至从异常处理向量表中取出的地址，开始执行程序。此时的转移不是延迟转移。

3.7.3 槽非法指令

将紧接着延迟转移指令之后配置的指令称为 “配置给延迟槽的指令”。配置给延迟槽的指令为未定义代码时，如果对此未定义的代码进行解码，就开始槽非法指令的异常处理；配置给延迟槽的指令为改写 PC 指令时，如果对改写 PC 指令进行解码，就开始槽非法指令的异常处理。并且，在无 FPU 的产品及有 FPU 的产品且 FPU 为模块待机状态时，将与浮点指令及 FPU 相关的 CPU 指令作为未定义代码处理，如果配置给延迟槽，当该指令被解码时就开始槽非法指令异常处理。

另外，无寄存器存储体的产品的寄存器存储体相关指令也作为未定义代码处理，如果配置给延迟槽，当该指令被解码时，开始槽非法指令异常处理。

并且，配置给延迟槽的指令是 32 位指令、 RESBANK 指令、 DIVS 指令及 DIVU 指令时，该指令被解码时也开始槽非法指令异常处理。

槽非法指令异常处理时， CPU 的运行如下：

(1) 从异常处理向量表中取出异常服务程序的起始地址。
(2) 将状态寄存器 （SR）保存到堆栈。
(3) 将程序计数器 （PC）保存到堆栈。保存的PC值是紧接着未定义代码、改写PC指令、 32位指令、RESBANK指令、 DIVS指令及DIVU指令之前的延迟转移指令的转移目标地址。
(4) 转移至从异常处理向量表中取出的地址，开始执行程序。此时的转移不是延迟转移。

3.7.4 一般非法指令

如果对配置在紧接着延迟转移指令 （延迟槽）以外的未定义代码进行解码，就开始一般非法指令的异常处理。另外，在无 FPU 的产品及有 FPU 的产品且为模块待机状态时，将与浮点指令及 FPU 相关的 CPU 指令作为未定义代码处理，如果配置在紧接着延迟转移指令 （延迟槽）以外的情况时，如对该指令进行解码就开始一般非法指令异常处理。

另外，无寄存器存储体的产品的寄存器存储体相关指令也作为未定义代码处理，如果配置在紧接着延迟转移指令 （延迟槽）以外的情况时，如对该指令进行解码就开始槽非法指令的异常处理。

一般非法指令异常处理时， CPU 按照与槽非法指令异常处理相同的步骤运行。但是，和槽非法指令的异常处理不同，保存的 PC 值是此未定义代码的起始地址。

3.7.5 整数除法指令

如果整数除法指令执行被零除时，或者整数除法的结果上溢时，发生整数除法异常。被零除异常的源指令是 DIVU 和 DIVS。上溢异常的源指令只有 DIVS，且只有在用 –1 除负的最大值时发生。发生整数除法异常时， CPU 的运行如下：

(1) 从异常处理向量表中取出对应发生的整数除法异常的异常服务程序的起始地址。
(2) 将状态寄存器 （SR）保存到堆栈。
(3) 将程序计数器 （PC）保存到堆栈。保存的PC值是发生异常的整数除法指令的起始地址。
(4) 转移至从异常处理向量表中取出的地址，开始执行程序。此时的转移不是延迟转移。

3.7.6 浮点运算指令

FPSCR 寄存器的允许字段中的 V、 Z、 O、 U 或 I 位被置位时，发生 FPU 异常。表示浮点运算指令引起了 IEEE754 规格中定义的无效运算异常、被零除异常、上溢 （有可能的指令）、下溢 （有可能的指令）以及不正确异常 （有可能的指令）。

成为异常源的浮点运算指令如下：

FADD、FSUB、FMUL、FDIV、FMAC、FCMP/EQ、FCMP/GT、FLOAT、FTRC、FCNVDS、FCNVSD、FSQRT

只有对应的允许位置位时，才发生 FPU 异常。 FPU 检测出异常源时，停止 FPU 的运行并通知 CPU 异常发生。 CPU 开始异常处理的运行如下：

(1) 从异常处理向量表中取出储存在VBR+H'00000034里的异常服务程序的起始地址。
(2) 将状态寄存器 （SR）保存到堆栈。
(3) 将程序计数器 （PC）保存到堆栈。保存的PC值是最后已执行指令的下一条指令的起始地址。
(4) 转移至储存在VBR+H'00000034的地址。

FPSCR 的异常标志位无论 FPU 异常是否被接受，都会被更新，用户用明示性地指令清除前，将一直保持置位。每次执行 FPU 指令， FPSCR 的源位就会改变。

另外， FPSCR 寄存器的允许字段中的 V 位被置位，并且 FPSCR 的 QIS 位被置位时，如果输入 qNaN 或者±∞到浮点运算指令源，就开始 FPU 异常处理。

3.8 不接受异常处理的情况

如表 3.9 所示，地址错误、 RAM 错误、 FPU 异常、寄存器存储体错误 （上溢）以及中断，在紧接延迟转移指令之后发生时，有可能不立即被接受而被保留。此时，当能够接受异常的指令被解码时接受。

表 3.9 发生紧接延迟转移指令之后的异常源

发生时间	异常源				
	地址错误	RAM 错误	FPU 异常	寄存器存储体错误 （上溢）	中断
紧接延迟转移指令 * 之后	×	×	×	×	×

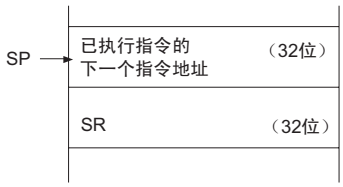
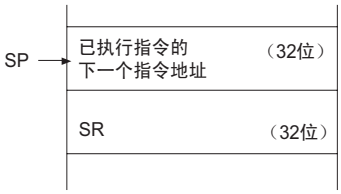
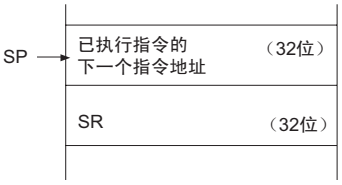
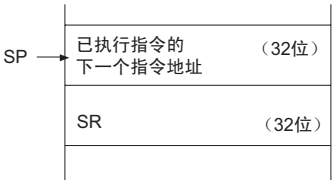
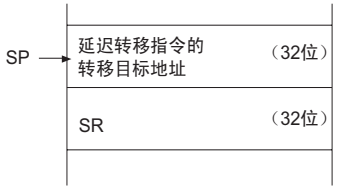
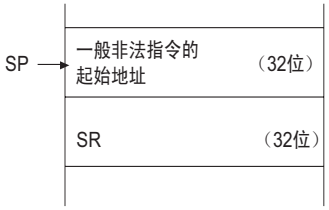
【注】 ×：不被接受

* 延迟转移指令：JMP、 JSR、 BRA、 BSR、 RTS、 RTE、 BF/S、 BT/S、 BSRF、 BRAF

3.9 异常处理后的堆栈状态

异常处理结束后的堆栈状态如表 3.10 所示。

表 3.10 异常处理结束后的堆栈状态

种类	堆栈状态	种类	堆栈状态
地址错误		中断	
RAM 错误		寄存器存储体错误 （上溢）	
寄存器存储体错误（下溢）		整数除法指令（被 0 除，上溢）	
陷阱指令		槽非法指令	
一般非法指令		FPU 异常	

3.10 使用时的注意事项

3.10.1 堆栈指针 （SP）的值

SP 的值必须是 4 的倍数。否则，如果在异常处理中堆栈被存取，就会发生地址错误。

3.10.2 向量基址寄存器 （VBR）的值

VBR 的值必须是 4 的倍数。否则，如果在异常处理中向量被存取，就会发生地址错误。

3.10.3 在地址错误异常处理的堆栈存取时发生的地址错误

如果 SP 不是 4 的倍数，就会在异常处理 （中断等）的堆栈存取时发生地址错误，在该异常处理结束后转移到地址错误异常处理。在地址错误异常处理的堆栈存取时也会发生地址错误。但是，为了不使地址错误异常处理的堆栈存取无限继续，不接受此时的地址错误。因此，能将程序的控制转移到地址错误的异常服务程序并进行错误处理。

如果在异常处理的堆栈存取时发生地址错误，就执行堆栈存取的总线周期 （写）。在 SR 和 PC 的堆栈存取时，因为 SR 和 PC 的 SP 分别减 4，所以即使在堆栈存取结束后， SP 的值也不是 4 的倍数。另外，堆栈存取时输出的地址值是 SP 的值，输出发生错误的地址。此时，堆栈存取的写数据不定。

第 4 章 指令的特点

4.1 RISC 方式

指令为 RISC 方式，特点如下：

(1) 16位固定长指令

基本指令为16位固定长。据此能提高程序的编码效率。

(2) 追加32位固定长指令

在SH-2A/SH2A-FPU中追加32位固定长的指令。据此提高性能及使用的便利性。

(3) 1条指令/1个状态

采用流水线方式，基本指令能够以一条指令一个状态执行。

(4) 数据长度

运算的基本数据长度为长字。存储器的存取长度能选择字节/字/长字。存储器的字节和字数据，在进行符号扩展后用长字进行运算。立即数在算术运算时进行符号扩展，而在逻辑运算时进行零扩展后，用长字进行运算。

表 4.1 字数据的符号扩展

SH-2A/SH2A-FPU CPU	说明	其它 CPU 的例子
MOV.W @(disp、 PC),R1 ADD R1,R0 . . . . . . .DATA.W H'1234	R1 经符号扩展为 32 位，变为 H'00001234。 然后用 ADD 指令进行运算。	ADD.W #H'1234,R0

【注】 通过 @(disp、 PC) 参照立即数。

(5) 加载存储结构

在寄存器间进行基本运算。将数据加载至寄存器后，与存储器进行运算 （加载存储结构）。但操作AND等位的指令直接对存储器进行运算。

(6) 延迟转移

除去一部分指令后的无条件转移指令是延迟转移指令。延迟转移指令时先执行紧接着延迟指令之后的指令，然后进行转移。因此，减少了转移时的流水线混乱现象。

延迟转移时，转移动作本身发生在槽指令执行之后，但指令的执行 （寄存器的更新等）仍然按延迟转移指令→延迟槽指令的顺序进行。例如，即使在延迟槽中更改保存了转移地址的寄存器，转移地址仍然是更改前的寄存器内容。

表 4.2 延迟转移指令

SH-2A/SH2A-FPU CPU	说明	其它 CPU 的例子
BRA TRGET ADD R1,R0	转移到 TRGET 前执行 ADD。	ADD.W R1,R0 BRA TRGET

(7) 追加无延迟槽无条件转移指令

在SH-2A/SH2A-FPU，追加不执行延迟槽指令的无条件转移指令。因此，可以削减无用的NOP指令，并能削减代码长度。

(8) 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的乘法和32×32+64→64的乘法累加运算。

(9) T位

比较结果反映在状态寄存器 （SR）的T位，根据真假进行条件转移。只用所需最少限度的指令改变T位，提高了处理速度。

表 4.3 T 位

SH-2A/SH2A-FPU CPU	说明	其它 CPU 的例子
CMP/GE R1,R0 BT TRGET0 BF TRGET1	当 R0 ≥ R1 时， T 位被置位。 当 R0 ≥ R1 时，转移到 TRGET0。 当 R0 < R1 时，转移到 TRGET1。	CMP.W R1,R0 BGE TRGET0 BLT TRGET1
ADD #–1,R0 CMP/EQ #0,R0 BT TRGET	ADD 时不改变 T 位。 当 R0=0 时， T 位被置位。 当 R0=0 时，进行转移。	SUB.W #1,R0 BEQ TRGET

(10) 立即数

字节的立即数分配在操作码中，字和长字的立即数不分配在操作码中，而分配在存储器的表中。通过使用了带位移量的PC相对寻址方式的立即数传送指令 （MOV），参照存储器的表。

另外，在SH-2A/SH2A-FPU， 17～28位的立即数也可分配在操作码中。但是， 21～28位的立即数在寄存器传送后，需执行OR指令。

表 4.4 通过立即数参照

区分	SH-2A/SH2A-FPU CPU	其它 CPU 的例子
8 位立即数	MOV H'12,R0	MOV.B #H'12,R0
16 位立即数	MOVI20 #H'1234,R0	MOV.W #H'1234,R0
20 位立即数	MOVI20 #H'12345,R0	MOV.L #H'12345,R0
28 位立即数	MOVI20S #H'12345, R0 OR #H'67,R0	MOV.L #H'1234567,R0
32 位立即数	MOV.L @(disp、 PC),R0 ・・・・・・ .DATA.L H'12345678	MOV.L #H'12345678,R0

【注】 用 @(disp、 PC) 参照立即数。

(11) 绝对地址

通过绝对地址参照数据时，预先将绝对地址的值分配到存储器的表中。执行指令时，通过加载立即数的方法将该值传送到寄存器，并以寄存器间接寻址模式参照数据。

另外，在SH-2A/SH2A-FPU，通过小于等于28位的绝对地址参照数据时，可以将分配在操作码中的立即数传送到寄存器，以寄存器间接寻址模式参照数据。但是通过21～28位的绝对地址参照数据时，寄存器传送后，需要使用OR指令。

表 4.5 通过绝对地址参照

区分	SH-2A/SH2A-FPU CPU	其它 CPU 的例子
小于等于 20 位	MOVI20 #H'12345,R1 MOV.B @R1,R0	MOV.B @H'12345,R0
21 ～ 28 位	MOVI20S #H'12345,R1 OR #H'67,R1 MOV.B @R1, R0	MOV.B @H'1234567,R0
大于等于 29 位	MOV.L @(disp,PC)、 R1 MOV.B @R1,R0 ・・・・・・ .DATA.L H'12345678	MOV.B @H'12345678,R0

(12) 16位/32位的位移量

通过16位或者32位的位移量参照数据时，预先将位移量的值分配到存储器的表中。执行指令时，通过加载立即数的方法将该值传送到寄存器，并以带变址的寄存器间接寻址模式参照数据。

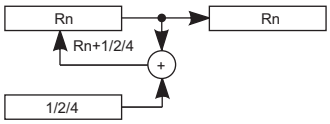
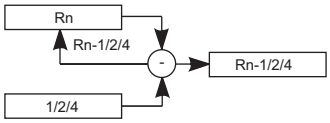
表 4.6 通过位移量参照

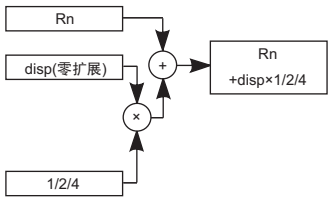
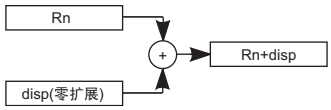
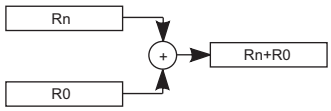
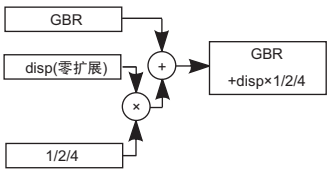
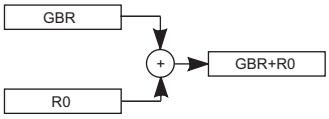
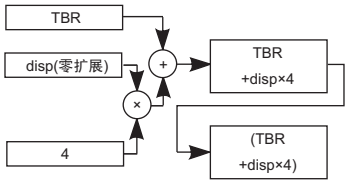
区分	SH-2A/SH2A-FPU CPU	其它 CPU 的例子
16 位的位移量	MOV.W @(disp、 PC),R0 MOV.W @(R0、 R1),R2 . . . . . . DATA.W H'1234	MOV.W @(H'1234,R1)、 R2

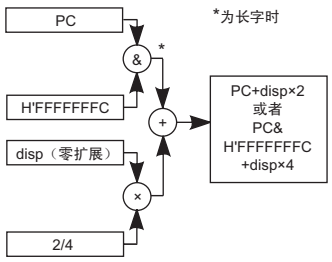
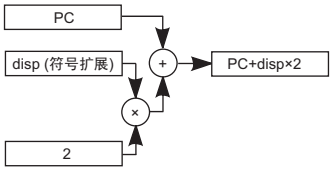
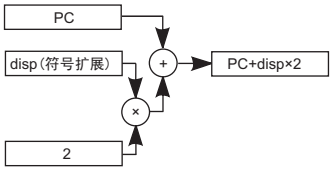
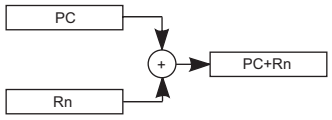
4.2 寻址模式

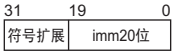
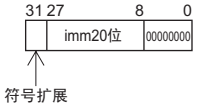
寻址方式和有效地址的计算方法如下所示。

表 4.7 寻址模式与有效地址

寻址模式	指令格式	有效地址的计算方法	计算式
寄存器直接	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	@Rn	有效地址为寄存器 Rn 的内容。 	Rn
后增寄存器间接	@Rn+	有效地址为寄存器 Rn 的内容。执行指令后，给 Rn 加上常数。操作数长度为字节时，常数为 1、操作数长度为字时，常数为 2、操作数长度为长字时，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	@-Rn	有效地址为先减去常数后的寄存器 Rn 的内容。操作数为字节时，常数为 1、操作数为字时，常数为 2、操作数为长字时，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）

寻址模式	指令格式	有效地址的计算方法	计算式
带位移量的寄存器间接	@(disp:4,Rn)	有效地址为寄存器 Rn 加上 4 位位移量 disp 后的内容。 disp 进行零扩展后，操作数长度为字节时乘 1，为字时乘 2，为长字时乘 4。 	字节：Rn+disp 字：Rn+disp×2 长字：Rn+disp×4
	@(disp:12,Rn)	有效地址为寄存器 Rn 加上 12 位位移量 disp 后的内容。 disp 进行零扩展。 	字节：Rn+disp 字：Rn+disp 长字：Rn+disp
带变址的寄存器间接	@(R0,Rn)	有效地址为寄存器 Rn 加上 R0 后的内容。 	Rn+R0
带位移量的 GBR 间接	@(disp:8,GBR)	有效地址为寄存器 GBR 加上 8 位位移量 disp 后的内容。 disp 进行零扩展后，操作数长度为字节时乘 1，为字时乘 2，为长字时乘 4。 	字节：GBR+disp 字：GBR+disp×2 长字：GBR+disp×4
带变址的 GBR 间接	@(R0,GBR)	有效地址为寄存器 GBR 加上 R0 后的内容。 	GBR+R0
带位移量的 TBR 双重间接	@@ (disp:8,TBR)	有效地址为寄存器 TBR 加上 8 位位移量 disp 后地址的内容。 disp 进行零扩展后乘 4。 	（TBR+disp×4）地址的内容

寻址模式	指令格式	有效地址的计算方法	计算式
带位移量的 PC 相对	@(disp:8,PC)	有效地址为寄存器 PC 加上 8 位位移量 disp 后的内容。 disp 在进行零扩展后，操作数长度为字时乘 2，为长字时乘 4。另外，为长字时屏蔽 PC 的低 2 位。 	字：PC+disp×2 长字： PC&H'FFFFFFFC+disp×4
PC 相对	disp:8	有效地址为寄存器 PC 加上 2 倍符号扩展 8 位位移量 disp 后的内容。 	PC+disp×2
	disp:12	有效地址为寄存器 PC 加上 2 倍符号扩展 12 位位移量 disp 后的内容。 	PC+disp×2
	Rn	有效地址为寄存器 PC 加上 Rn 后的内容。 	PC+Rn

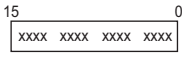
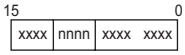
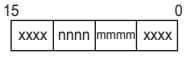
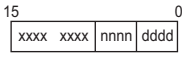
寻址模式	指令格式	有效地址的计算方法	计算式
立即数	#imm:20	MOVI20 指令的 20 位立即数 imm 进行符号扩展。 	—
		MOVI20S 指令的 20 位立即数 imm 向左移 8 位，高位进行符号扩展，低位进行零填充。 	—
	#imm:8	TST、 AND、 OR、 XOR 指令的 8 位立即数 imm 进行零扩展。	—
	#imm:8	MOV、ADD、CMP/EQ 指令的 8 位立即数 imm 进行符号扩展。	—
	#imm:8	TRAPA 指令的 8 位立即数 imm 进行零扩展后乘 4。	—
	#imm:3	BAND、 BOR、 BXOR、 BST、 BLD、 BSET、 BCLR 指令的 3 位立即数 imm 表示位的位置。	—

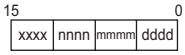
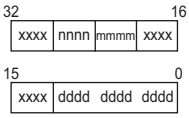
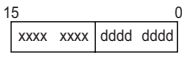
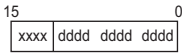
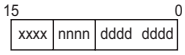
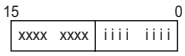
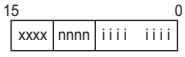
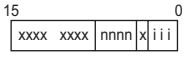
4.3 指令格式

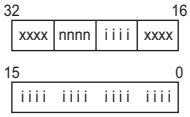
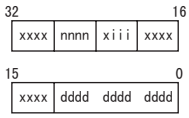
表示指令格式、源操作数及目标操作数。操作数的含义因操作码不同而不同，符号如下所示：

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	指令例子	
0 格式		—	—	NOP	
n 格式		—	nnnn：寄存器直接	MOV T	Rn
		控制寄存器或者系统寄存器	nnnn：寄存器直接	STS	MACH,Rn
		R0 （寄存器直接）	nnnn：寄存器直接	DIVU	R0, Rn
		控制寄存器或者系统寄存器	nnnn：先减寄存器间接	STC.L	SR,@-Rn
		mmmm：寄存器直接	R15 （先减寄存器间接）	MOVMU.L	Rm, @-R15
		R15 （后增寄存器间接）	nnnn：寄存器直接	MOVMU.L	@R15+,Rn
		R0 （寄存器直接）	nnnn：后增寄存器间接	MOV.L	R0,@Rn+
m 格式		mmmm：寄存器直接	控制寄存器或者系统寄存器	LDC	Rm,SR
		mmmm：后增寄存器间接	控制寄存器或者系统寄存器	LDC	@Rm+,SR
		mmmm：寄存器间接	—	JMP	@Rm
		mmmm：先减寄存器间接	R0 （寄存器直接）	MOV.L	@-Rm,R0
		mmmm：使用 Rm 的 PC 相对	—	BRAF	Rm
nm 格式		mmmm：寄存器直接	nnnn：寄存器直接	ADD	Rm,Rn
		mmmm：寄存器直接	nnnn：寄存器间接	MOV.L	Rm,@Rn
		mmmm：后增寄存器间接 （乘法累加运算） nnnn*：后增寄存器间接 （乘法累加运算）	MACH,MACL	MAC.W	@Rm+,@Rn+
		mmmm：后增寄存器间接	nnnn：寄存器直接	MOV.L	@Rm+,Rn
		mmmm：寄存器直接	nnnn：先减寄存器间接	MOV.L	Rm,@-Rn
		mmmm：寄存器直接	nnnn：带变址的寄存器间接	MOV.L	Rm,@(R0,Rn)
md 格式		mmmmdddd：带位移量寄存器间接	R0 （寄存器直接）	MOV.B	@(disp,Rm),R0
nd4 格式		R0 （寄存器直接）	nnnndddd：带位移量寄存器间接	MOV.B	R0,@(disp,Rn)

指令格式		源操作数	目标操作数	指令例子
nmd 格式		mmmm：寄存器直接	nnnndddd：带位移量寄存器间接	MOV.L Rm,@(disp,Rn)
		mmmmdddd：带位移量寄存器间接	nnnn：寄存器直接	MOV.L @(disp、 Rm),Rn
nmd12 格式		mmmm：寄存器直接	nnnndddd：带位移量寄存器间接	MOV.L Rm、@(disp12,Rn)
		mmmmdddd：带位移量寄存器间接	nnnn：寄存器直接	MOV.L @(disp12, Rm), Rn
d 格式		dddddddd：带位移量的 GBR 间接	R0 （寄存器直接）	MOV.L @(disp,GBR),R0
		R0 （寄存器直接）	dddddddd：带位移量的 GBR 间接	MOV.L R0,@(disp,GBR)
		dddddddd：带位移量的 PC 相对	R0 （寄存器直接）	MOVA @(disp,PC),R0
		dddddddd：带位移量的 TBR 双重间接	—	JSR/N @@(disp8,TBR)
		dddddddd：PC 相对	—	BF label
d12 格式		ddddddddddddd：PC 相对	—	BRA label (label=disp+PC)
nd8 格式		dddddddd：带位移量的 PC 相对	nnnn：寄存器直接	MOV.L @(disp,PC),Rn
i 格式		iiiiiiii：立即	带变址的 GBR 间接	AND.B #imm,@(R0,GBR)
		iiiiiiii：立即	R0 （寄存器直接）	AND #imm,R0
		iiiiiiii：立即	—	TRAPA #imm
ni 格式		iiiiiiii：立即	nnnn：寄存器直接	ADD #imm,Rn
ni3 格式		nnnn：寄存器直接 iii：立即	—	BLD #imm3,Rn
		—	nnnn：寄存器直接 iii：立即	BST #imm3,Rn

指令格式		源操作数	目标操作数	指令例子
ni20 格式		iiiiiiiiiiiiiiiiiiii： 立即	nnnn：寄存器直接	MOVI20 #imm20, Rn
nid 格式		nnnnddddddddddddd：带位移量的寄存器间接 iii：立即	—	BLD.B #imm3,@(disp12,Rn)
		—	nnnnddddddddddddd： 带位移量的寄存器间接 iii：立即	BST.B #imm3,@(disp12,Rn)

【注】 * 在乘加指令中 nnnn 为源寄存器。

第 5 章 指令系统

5.1 指令系统的分类顺序

指令系统的分类顺序如表 5.1 所示。

表 5.1 指令的分类

分类	指令的种类	操作码	功能	指令数
数据传送指令	13	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 传送反向堆栈	62
		MOVA	传送执行地址	
		MOVI20	传送 20 位立即数	
		MOVI20S	传送 20 位立即数 左移 8 位	
		MOVML	R0 ～ Rn 的寄存器保存 / 恢复	
		MOVMU	Rn ～ R14、 PR 的寄存器保存 / 恢复	
		MOVRT	对 T 位取反后传送到 Rn	
		MOVT	传送 T 位	
		MOVU	传送无符号数据	
		NOTT	T 位取反	
		PREF	预取操作数高速缓存	
		SWAP	转换高位和低位	
		XTRCT	抽出连接寄存器的中间部分	
算术运算指令	26	ADD	2 进制加法	40
		ADDC	带进位的 2 进制加法	
		ADDV	带上溢的 2 进制加法	
		CMP/cond	比较	
		CLIPS	带符号的饱和值比较	
		CLIPU	无符号的饱和值比较	
		DIVS	带符号除法 (32÷32)	
		DIVU	无符号除法 (32÷32)	
		DIV1	单步除法	
		DIV0S	带符号的单步除法初始化	
		DIV0U	无符号的单步除法初始化	
		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	
		MULR	带 Rn 结果保存符号乘法	
		MULS	带符号乘法	
		MULU	无符号乘法	
		NEG	符号取反	

分类	指令的种类	操作码	功能	指令数
算术运算指令	26	NEGC	带借位的符号取反	40
		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	
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	逻辑异或运算	
移位指令	12	ROTL	左循环 1 位	16
		ROTR	右循环 1 位	
		ROTCL	带 T 位的左循环 1 位	
		ROTCR	带 T 位的右循环 1 位	
		SHAD	动态算术移位	
		SHAL	算术左移 1 位	
		SHAR	算术右移 1 位	
		SHLD	动态逻辑移位	
		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	10	BF	条件转移、带延迟的条件转移 （当 T=0 时进行转移）	15
		BT	条件转移、带延迟的条件转移 （当 T=1 时进行转移）	
		BRA	带延迟的无条件转移	
		BRAF	带延迟的无条件转移	
		BSR	转移到带延迟的子程序过程	
		BSRF	转移到带延迟的子程序过程	
		JMP	带延迟的无条件转移	
		JSR	转移到子程序过程、 转移到带延迟的子程序过程	
		RTS	从子程序过程返回、 从带延迟的子程序过程返回	
		RTV/N	从带 Rm→R0 传送的子程序过程返回	
系统控制指令	14	CLRT	清除 T 位	36
		CLRMAC	清除 MAC 寄存器	
		LDBANK	从指定寄存器存储体入口的寄存器返回	
		LDC	加载到控制寄存器	
		LDS	加载到系统寄存器	
		NOP	无操作	
		RESBANK	从寄存器存储体的寄存器返回	
		RTE	从异常处理返回	
		SETT	T 位的置位	

分类	指令的种类	操作码	功能	指令数
系统控制指令	14	SLEEP	转移到低功耗状态	36
		STBANK	将寄存器保存到指定寄存器存储体入口	
		STC	从控制寄存器的存储	
		STS	从系统寄存器的存储	
		TRAPA	陷阱异常处理	
浮点运算指令	19	FABS	浮点数绝对值	48
		FADD	浮点数加法	
		FCMP	浮点数比较	
		FCNVDS	从双精度转换为单精度	
		FCNVSD	从单精度转换为双精度	
		FDIV	浮点数除法	
		FLDI0	浮点数加载立即数 0	
		FLDI1	浮点数加载立即数 1	
		FLDS	将浮点数加载到系统寄存器 FPUL	
		FLOAT	将整数转换为浮点数	
		FMAC	浮点数乘法累加运算	
		FMOV	浮点数传送	
		FMUL	浮点数乘法	
		FNEG	浮点数符号取反	
		FSCHG	SZ 位取反	
		FSQRT	浮点数平方根	
		FSTS	从系统寄存器 FPUL 存储浮点数	
		FSUB	浮点数减法	
		FTRC	取浮点数的整数	
有关 FPU 的 CPU 指令	2	LDS	加载到浮点系统寄存器	8
		STS	从浮点系统寄存器的存储	
位操作指令	10	BAND	位与	14
		BCLR	位清除	
		BLD	位加载	
		BOR	位或	
		BSET	置位	
		BST	保存位	
		BXOR	位异或	
		BANDNOT	位与非	
		BORNOT	位或非	
		BLDNOT	位非加载	
	计 112			253

通过以下格式按照分类顺序说明指令的操作码、运行和执行状态。

指令	操作码	操作概略	执行状态	T 位
用助记符表示。 符号说明 Rm ： 源寄存器 Rn ： 目标寄存器 imm ： 立即数 disp ： 位移量 *2	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	表示运行的概略 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	是无等待时的值。 *1	表示执行指令后的 T 位的值。 符号说明 — ： 不变化

【注】 *1 有关指令的执行状态

表中所示的执行状态为最少值。实际上根据以下条件，指令执行的状态数会增加。

（1）当取指令和数据存取发生竞争时

（2）当加载指令 （存储器→寄存器）的目标寄存器与紧接着的指令所使用的寄存器相同时

*2 根据指令的操作数长度等缩放 （×1、 ×2、 ×4）。

详细内容请参照 “第6章 各指令的说明”。

5.1.1 数据传送指令

表 5.2 数据传送指令

指令		操作码	运行	执行状态	T位	兼容性		
						SH2E	SH4	新 SH-2A/ SH2A-FPU
MOV	#imm,Rn	1110nnnniiiiiiii	imm→ 符号扩展 →Rn	1	—	○	○	
MOV.W	@(disp,PC), Rn	1001nnnndddddddd	(disp×2+PC)→ 符号扩展 →Rn	1	—	○	○	
MOV.L	@(disp,PC), Rn	1101nnnndddddddd	(disp×4+PC)→Rn	1	—	○	○	
MOV	Rm,Rn	0110nnnnmmmm0011	Rm→Rn	1	—	○	○	
MOV.B	Rm,@Rn	0010nnnnmmmm0000	Rm→(Rn)	1	—	○	○	
MOV.W	Rm,@Rn	0010nnnnmmmm0001	Rm→(Rn)	1	—	○	○	
MOV.L	Rm,@Rn	0010nnnnmmmm0010	Rm→(Rn)	1	—	○	○	
MOV.B	@Rm,Rn	0110nnnnmmmm0000	(Rm)→ 符号扩展 →Rn	1	—	○	○	
MOV.W	@Rm,Rn	0110nnnnmmmm0001	(Rm)→ 符号扩展 →Rn	1	—	○	○	
MOV.L	@Rm,Rn	0110nnnnmmmm0010	(Rm)→Rn	1	—	○	○	
MOV.B	Rm,@- Rn	0010nnnnmmmm0100	Rn-1→Rn、 Rm→(Rn)	1	—	○	○	
MOV.W	Rm,@- Rn	0010nnnnmmmm0101	Rn-2→Rn、 Rm→(Rn)	1	—	○	○	
MOV.L	Rm,@- Rn	0010nnnnmmmm0110	Rn-4→Rn、 Rm→(Rn)	1	—	○	○	
MOV.B	@Rm+,Rn	0110nnnnmmmm0100	(Rm)→ 符号扩展 →Rn,Rm+1→Rm	1	—	○	○	
MOV.W	@Rm+,Rn	0110nnnnmmmm0101	(Rm)→ 符号扩展 →Rn,Rm+2→Rm	1	—	○	○	
MOV.L	@Rm+,Rn	0110nnnnmmmm0110	(Rm)→Rn,Rm+4→Rm	1	—	○	○	
MOV.B	R0,@(disp, Rn)	10000000nnnndddd	R0→(disp+Rn)	1	—	○	○	
MOV.W	R0,@(disp, Rn)	10000001nnnndddd	R0→(disp×2+Rn)	1	—	○	○	
MOV.L	Rm,@(disp,Rn)	0001nnnnmmmmdddd	Rm→(disp×4+Rn)	1	—	○	○	
MOV.B	@(disp,Rm),R0	10000100mmmmdddd	(disp+Rm)→ 符号扩展 →R0	1	—	○	○	
MOV.W	@(disp,Rm), R0	10000101mmmmdddd	(disp×2+Rm)→ 符号扩展 →R0	1	—	○	○	
MOV.L	@(disp,Rm), Rn	0101nnnnmmmmdddd	(disp×4+Rm)→Rn	1	—	○	○	
MOV.B	Rm,@(R0,Rn)	0000nnnnmmmm0100	Rm→(R0+Rn)	1	—	○	○	
MOV.W	Rm,@(R0,Rn)	0000nnnnmmmm0101	Rm→(R0+Rn)	1	—	○	○	
MOV.L	Rm,@(R0,Rn)	0000nnnnmmmm0110	Rm→(R0+Rn)	1	—	○	○	
MOV.B	@(R0,Rm),Rn	0000nnnnmmmm1100	(R0+Rm)→ 符号扩展 →Rn	1	—	○	○	
MOV.W	@(R0,Rm),Rn	0000nnnnmmmm1101	(R0+Rm)→ 符号扩展 →Rn	1	—	○	○	
MOV.L	@(R0,Rm),Rn	0000nnnnmmmm1110	(R0+Rm)→Rn	1	—	○	○	
MOV.B	R0,@(disp,GBR)	11000000dddddddd	R0→(disp+GBR)	1	—	○	○	
MOV.W	R0,@(disp,GBR)	11000001dddddddd	R0→(disp×2+GBR)	1	—	○	○	
MOV.L	R0,@(disp,GBR)	11000010dddddddd	R0→(disp×4+GBR)	1	—	○	○	
MOV.B	@(disp,GBR), R0	11000100dddddddd	(disp+GBR)→ 符号扩展 →R0	1	—	○	○	
MOV.W	@(disp,GBR), R0	11000101dddddddd	(disp×2+GBR)→ 符号扩展 →R0	1	—	○	○	
MOV.L	@(disp,GBR),R0	11000110dddddddd	(disp×4+GBR)→R0	1	—	○	○	
MOV.B	R0,@Rn+	0100nnnn10001011	R0→(Rn),Rn+1→Rn	1	—			○

指令		操作码	运行	执行状态	T 位	兼容性		
						SH2E	SH4	新 SH-2A/SH2A-FPU
MOV.W	R0,@Rn+	0100nnnn10011011	R0→(Rn),Rn+2→Rn	1	—			○
MOV.L	R0,@Rn+	0100nnnn10101011	R0→(Rn),Rn+4→Rn	1	—			○
MOV.B	@-Rm,R0	0100mmmm11001011	Rm-1→Rm, (Rm)→ 符号扩展 →R0	1	—			○
MOV.W	@-Rm,R0	0100mmmm11011011	Rm-2→Rm、 (Rm)→ 符号扩展 →R0	1	—			○
MOV.L	@-Rm,R0	0100mmmm11101011	Rm-4→Rm,(Rm)→R0	1	—			○
MOV.B	Rm, @(disp12,Rn)	0011nnnnmmmm0001 0000dddddddddddd	Rm→(disp+Rn)	1	—			○
MOV.W	Rm,@(disp12,Rn)	0011nnnnmmmm0001 0001dddddddddddd	Rm→(disp×2+Rn)	1	—			○
MOV.L	Rm@(disp12,Rn)	0011nnnnmmmm0001 0010dddddddddddd	Rm→(disp×4+Rn)	1	—			○
MOV.B	@(disp12,Rm),Rn	0011nnnnmmmm0001 0100dddddddddddd	(disp+Rm)→ 符号扩展 →Rn	1	—			○
MOV.W	@(disp12,Rm),Rn	0011nnnnmmmm0001 0101dddddddddddd	(disp×2+Rm)→ 符号扩展 →Rn	1	—			○
MOV.L	@(disp12,Rm),Rn	0011nnnnmmmm0001 0110dddddddddddd	(disp×4+Rm)→Rn	1	—			○
MOVA	@(disp,PC),R0	11000111dddddddd	disp×4+PC→R0	1	—	○	○	
MOVI20	#imm20,Rn	0000nnnniiii0000 iiiiiiiiiiiiiiii	imm→ 符号扩展 →Rn	1	—			○
MOVI20S	#imm20,Rn	0000nnnniiii0001 iiiiiiiiiiiiiiii	imm<<8→ 符号扩展 →Rn	1	—			○
MOVML.L	Rm,@-R15	0100mmmm11110001	R15-4→R15,Rm→(R15) R15-4→R15,Rm-1→(R15) : R15-4→R15,R0→(R15) ※ Rm=R15 时， Rm 读作 PR	1 ～ 16	—			○
MOVML.L	@R15+,Rn	0100nnnn11110101	(R15)→R0、 R15+4→R15 (R15)→R1、 R15+4→R15 : (R15)→Rn ※ Rn=R15 时， Rn 读作 PR	1 ～ 16	—			○
MOVMU.L	Rm,@-R15	0100mmmm11110000	R15-4→R15、 PR→(R15) R15-4→R15、 R14→(R15) : R15-4→R15,Rm→(R15) ※ Rm=R15 时， Rm 读作 PR	1 ～ 16	—			○
MOVMU.L	@R15+,Rn	0100nnnn11110100	(R15)→Rn、 R15+4→R15 (R15)→Rn+1、 R15+4→R15 : (R15)→R14、 R15+4→R15 (R15)→PR ※ Rn=R15 时， Rn 读作 PR	1 ～ 16	—			○

指令		操作码	运行	执行状态	T 位	兼容性		
						SH2E	SH4	新 SH-2A/SH2A-FPU
MOVRT	Rn	0000nnnn00111001	~T→Rn	1	—			○
MOVT	Rn	0000nnnn00101001	T→Rn	1	—	○	○	
MOVU.B	@(disp12,Rm),Rn	0011nnnnmmmm0001 1000dddddddddddd	(disp+Rm)→ 零扩展 →Rn	1	—			○
MOVU.W	@(disp12,Rm),Rn	0011nnnnmmmm0001 1001dddddddddddd	(disp×2+Rm)→ 零扩展 →Rn	1	—			○
NOTT		0000000001101000	~T→T	1	运算结果			○
PREF	@Rn	0000nnnn10000011	(Rn)→ 操作数高速缓存	1	—		○	
SWAP.B	Rm,Rn	0110nnnnmmmm1000	Rm→ 低位 2 字节的高低字节转换 →Rn	1	—	○	○	
SWAP.W	Rm,Rn	0110nnnnmmmm1001	Rm→ 高低字转换 →Rn	1	—	○	○	
XTRCT	Rm,Rn	0010nnnnmmmm1101	Rm:Rn 中间 32 位 →Rn	1	—	○	○	

5.1.2 算术运算指令

表 5.3 算术运算指令

指令		操作码	运行	执行状态	T 位	兼容性		
						SH2E	SH4	新 SH-2A/ SH2A-FPU
ADD	Rm,Rn	0011nnnnmmmm1100	Rn+Rm→Rn	1	—	○	○	
ADD	#imm,Rn	0111nnnniiiiiiii	Rn+imm→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	○	○	
CMP/EQ	#imm,R0	10001000iiiiiiii	R0=imm 时 1→T 其它时候 0→T	1	比较结果	○	○	
CMP/EQ	Rm,Rn	0011nnnnmmmm0000	Rn=Rm 时 1→T 其它时候 0→T	1	比较结果	○	○	
CMP/HS	Rm,Rn	0011nnnnmmmm0010	无符号 Rn ≥ Rm 时 1→T 其它时候 0→T	1	比较结果	○	○	
CMP/GE	Rm,Rn	0011nnnnmmmm0011	有符号 Rn ≥ Rm 时 1→T 其它时候 0→T	1	比较结果	○	○	
CMP/HI	Rm,Rn	0011nnnnmmmm0110	无符号 Rn>Rm 时 1→T 其它时候 0→T	1	比较结果	○	○	
CMP/GT	Rm,Rn	0011nnnnmmmm0111	有符号 Rn>Rm 时 1→T 其它时候 0→T	1	比较结果	○	○	
CMP/PL	Rn	0100nnnn00010101	Rn>0 时 1→T 其它时候 0→T	1	比较结果	○	○	
CMP/PZ	Rn	0100nnnn00010001	Rn ≥ 0 时 1→T 其它时候 0→T	1	比较结果	○	○	
CMP/STR	Rm,Rn	0010nnnnmmmm1100	当任意字节相等时 1→T 其它时候 0→T	1	比较结果	○	○	
CLIPS.B	Rn	0100nnnn10010001	Rn>(H'0000007F) 时， (H'0000007F) →Rn,1→CS Rn<(H'FFFFFF80) 时， (H'FFFFFF80) →Rn,1→CS	1	—			○
CLIPS.W	Rn	0100nnnn10010101	Rn>(H'00007FFF) 时， (H'00007FFF) →Rn,1→CS Rn<(H'FFFF8000) 时， (H'FFFF8000) →Rn,1→CS	1	—			○
CLIPU.B	Rn	0100nnnn10000001	Rn>(H'000000FF) 时， (H'000000FF) →Rn,1→CS	1	—			○
CLIPU.W	Rn	0100nnnn10000101	Rn>(H'0000FFFF) 时， (H'0000FFFF) →Rn,1→CS	1	—			○
DIV1	Rm,Rn	0011nnnnmmmm0100	单步除法 (Rn÷Rm)	1	计算结果	○	○	

指令		操作码	运行	执行状态	T 位	兼容性		
						SH2E	SH4	新 SH-2A/ SH2A-FPU
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	○	○	
DIVS	R0,Rn	0100nnnn10010100	带符号 Rn÷R0→Rn 32÷32→32 位	36	—			○
DIVU	R0,Rn	0100nnnn10000100	无符号 Rn÷R0→Rn 32÷32→32 位	34	—			○
DMULS.L	Rm,Rn	0011nnnnmmmm1101	带符号 Rn×Rm→MACH,MACL 32×32→64 位	2	—	○	○	
DMULU.L	Rm,Rn	0011nnnnmmmm0101	无符号 Rn×Rm→MACH,MACL 32×32→64 位	2	—	○	○	
DT	Rn	0100nnnn00010000	Rn － 1→Rn、 Rn 为 0 时 1→T 其它时候 0→T	1	比较结果	○	○	
EXTS.B	Rm,Rn	0110nnnnmmmm1110	从字节符号扩展 Rm→Rn	1	—	○	○	
EXTS.W	Rm,Rn	0110nnnnmmmm1111	从字符号进行扩展 Rm→Rn	1	—	○	○	
EXTU.B	Rm,Rn	0110nnnnmmmm1100	从字节符号进行零扩展 Rm→Rn	1	—	○	○	
EXTU.W	Rm,Rn	0110nnnnmmmm1101	从字符号进行零扩展 Rm→Rn	1	—	○	○	
MAC.L	@Rm+,@Rn+	0000nnnnmmmm1111	带符号 (Rn)×(Rm)+MAC→MAC 32×32+64→64 位	4	—	○	○	
MAC.W	@Rm+,@Rn+	0100nnnnmmmm1111	带符号 (Rn)×(Rm)+MAC→MAC 16×16+64→64 位	3	—	○	○	
MUL.L	Rm,Rn	0000nnnnmmmm0111	Rn×Rm→MACL 32×32→32 位	2	—	○	○	
MULR	R0,Rn	0100nnnn10000000	R0×Rn→Rn 32×32→32 位	2				○
MULS.W	Rm,Rn	0010nnnnmmmm1111	带符号 Rn×Rm→MACL 16×16→32 位	1	—	○	○	
MULU.W	Rm,Rn	0010nnnnmmmm1110	无符号 Rn×Rm→MACL 16×16→32 位	1	—	○	○	
NEG	Rm,Rn	0110nnnnmmmm1011	0-Rm→Rn	1	—	○	○	
NEGC	Rm,Rn	0110nnnnmmmm1010	0-Rm-T→Rn、借位 →T	1	借位	○	○	
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	上溢	○	○	

5.1.3 逻辑运算指令

表 5.4 逻辑运算指令

指令	操作码	运行	执行状态	T 位	兼容性		
					SH2E	SH4	新 SH-2A/ SH2A-FPU
AND Rm,Rn	0010nnnnmmmm1001	Rn & Rm→Rn	1	—	○	○	
AND #imm,R0	11001001iiiiiiii	R0 & imm→R0	1	—	○	○	
AND.B #imm,@(R0,GBR)	11001101iiiiiiii	(R0+GBR) & imm→(R0+GBR)	3	—	○	○	
NOT Rm,Rn	0110nnnnmmmm0111	~Rm→Rn	1	—	○	○	
OR Rm,Rn	0010nnnnmmmm1011	Rn ｜ Rm→Rn	1	—	○	○	
OR #imm,R0	11001011iiiiiiii	R0 ｜ imm→R0	1	—	○	○	
OR.B #imm,@(R0,GBR)	11001111iiiiiiii	(R0+GBR) ｜ imm→(R0+GBR)	3	—	○	○	
TAS.B @Rn	0100nnnn00011011	(Rn) 为 0 时 1→T，其它时候 0→T、 1→MSB of(Rn)	3	测试结果	○	○	
TST Rm,Rn	0010nnnnmmmm1000	Rn & Rm、结果为 0 时 1→T，否则 0→T	1	测试结果	○	○	
TST #imm,R0	11001000iiiiiiii	R0 & imm、结果为 0 时 1→T，否则 0→T	1	测试结果	○	○	
TST.B #imm,@(R0,GBR)	11001100iiiiiiii	(R0+GBR) & imm、结果为 0 时 1→T，否则 0→T	3	测试结果	○	○	
XOR Rm,Rn	0010nnnnmmmm1010	Rn ^ Rm→Rn	1	—	○	○	
XOR #imm,R0	11001010iiiiiiii	R0 ^ imm→R0	1	—	○	○	
XOR.B #imm,@(R0,GBR)	11001110iiiiiiii	(R0+GBR) ^ imm→(R0+GBR)	3	—	○	○	

5.1.4 移位指令

表 5.5 移位指令

指令	操作码	运行	执行状态	T 位	兼容性		
					SH2E	SH4	新 SH-2A/ SH2A-FPU
ROTL Rn	0100nnnn00000100	T←Rn←MSB	1	MSB	○	○	
ROTR Rn	0100nnnn00000101	LSB→Rn→T	1	LSB	○	○	
ROTCL Rn	0100nnnn00100100	T←Rn←T	1	MSB	○	○	
ROTCR Rn	0100nnnn00100101	T→Rn→T	1	LSB	○	○	
SHAD Rm,Rn	0100nnnnmmmm1100	Rm ≥ 0 时 Rn<<Rm→Rn Rm<0 时 Rn>>|Rm|→[MSB→Rn]	1	—		○	
SHAL Rn	0100nnnn00100000	T←Rn←0	1	MSB	○	○	
SHAR Rn	0100nnnn00100001	MSB→Rn→T	1	LSB	○	○	
SHLD Rm,Rn	0100nnnnmmmm1101	Rm ≥ 0 时 Rn<<Rm→Rn Rm<0 时 Rn>>|Rm|→[0→Rn]	1	—		○	
SHLL Rn	0100nnnn00000000	T←Rn←0	1	MSB	○	○	
SHLR Rn	0100nnnn00000001	0→Rn→T	1	LSB	○	○	
SHLL2 Rn	0100nnnn00001000	Rn<<2→Rn	1	—	○	○	
SHLR2 Rn	0100nnnn00001001	Rn>>2→Rn	1	—	○	○	
SHLL8 Rn	0100nnnn00011000	Rn<<8→Rn	1	—	○	○	
SHLR8 Rn	0100nnnn00011001	Rn>>8→Rn	1	—	○	○	
SHLL16 Rn	0100nnnn00101000	Rn<<16→Rn	1	—	○	○	
SHLR16 Rn	0100nnnn00101001	Rn>>16→Rn	1	—	○	○	

5.1.5 转移指令

表 5.6 转移指令

指令		操作码	运行	执行状态	T 位	兼容性		
						SH2E	SH4	新 SH-2A/ SH2A-FPU
BF	label	10001011dddddddd	T=0 时 disp×2+PC→PC, T=1 时 nop	3/1*	—	○	○	
BF/S	label	10001111dddddddd	延迟转移、 T=0 时 disp×2+PC→PC, T=1 时 nop	2/1*	—	○	○	
BT	label	10001001dddddddd	T=1 时 disp×2+PC→PC, T=0 时 nop	3/1*	—	○	○	
BT/S	label	10001101dddddddd	延迟转移、 T=1 时 disp×2+PC→PC, T=0 时 nop	2/1*	—	○	○	
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	—	○	○	
JMP	@Rm	0100mmmm00101011	延迟转移、 Rm→PC	2	—	○	○	
JSR	@Rm	0100mmmm00001011	延迟转移、 PC→PR,Rm→PC	2	—	○	○	
JSR/N	@Rm	0100mmmm01001011	PC-2→PR,Rm→PC	3	—			○
JSR/N	@@(disp8,TBR)	10000011dddddddd	PC-2→PR, (disp×4+TBR)→PC	5	—			○
RTS		0000000000001011	延迟转移、 PR→PC	2	—	○	○	
RTS/N		0000000001101011	PR→PC	3	—			○
RTV/N	Rm	0000mmmm01111011	Rm→R0,PR→PC	3	—			○

【注】 * 在不转移时为一个状态。

5.1.6 系统控制指令

表 5.7 系统控制指令

指令		操作码	运行	执行状态	T 位	兼容性		
						SH2E	SH4	新 SH-2A/ SH2A-FPU
CLRT		0000000000001000	0→T	1	0	○	○	
CLRMAC		0000000000101000	0→MACH,MACL	1	—	○	○	
LDBANK	@Rm,R0	0100mmmm11100101	（指定寄存器存储体入口）→R0	6	—			○
LDC	Rm,SR	0100mmmm00001110	Rm→SR	3	LSB	○	○	
LDC	Rm,TBR	0100mmmm01001010	Rm→TBR	1	—			○
LDC	Rm,GBR	0100mmmm00011110	Rm→GBR	1	—	○	○	
LDC	Rm,VBR	0100mmmm00101110	Rm→VBR	1	—	○	○	
LDC.L	@Rm+,SR	0100mmmm00000111	(Rm)→SR,Rm+4→Rm	5	LSB	○	○	
LDC.L	@Rm+,GBR	0100mmmm00010111	(Rm)→GBR,Rm+4→Rm	1	—	○	○	
LDC.L	@Rm+, VBR	0100mmmm00100111	(Rm)→VBR,Rm+4→Rm	1	—	○	○	
LDS	Rm,MACH	0100mmmm00001010	Rm→MACH	1	—	○	○	
LDS	Rm,MACL	0100mmmm00011010	Rm→MACL	1	—	○	○	
LDS	Rm,PR	0100mmmm00101010	Rm→PR	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	—	○	○	
NOP		0000000000001001	无操作	1	—	○	○	
RESBANK		0000000001011011	存储体 →R0 ～ R14,GBR,MACH,MACL,PR	9*	—			○
RTE		0000000000101011	延迟转移、堆栈区 →PC/SR	6	—	○	○	
SETT		0000000000011000	1→T	1	1	○	○	
SLEEP		0000000000011011	睡眠	5	—	○	○	
STBANK	R0,@Rn	0100nnnn11100001	R0→ （指定寄存器存储体入口）	7	—			○
STC	SR,Rn	0000nnnn00000010	SR→Rn	2	—	○	○	
STC	TBR,Rn	0000nnnn01001010	TBR→Rn	1	—			○
STC	GBR、 Rn	0000nnnn00010010	GBR→Rn	1	—	○	○	
STC	VBR,Rn	0000nnnn00100010	VBR→Rn	1	—	○	○	
STC.L	SR,@- Rn	0100nnnn00000011	Rn-4→Rn,SR→(Rn)	2	—	○	○	
STC.L	GBR,@- Rn	0100nnnn00010011	Rn-4→Rn,GBR→(Rn)	1	—	○	○	
STC.L	VBR,@- Rn	0100nnnn00100011	Rn-4→Rn,VBR→(Rn)	1	—	○	○	
STS	MACH,Rn	0000nnnn00001010	MACH→Rn	1	—	○	○	
STS	MACL,Rn	0000nnnn00011010	MACL→Rn	1	—	○	○	
STS	PR,Rn	0000nnnn00101010	PR→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	—	○	○	
TRAPA	#imm	11000011iiiiiiii	PC/SR→ 堆栈区，(imm×4+VBR)→PC	5	—	○	○	

【注】 有关指令的执行状态

表中所示的执行状态为最少值。实际上根据以下条件，指令执行的状态数会增加。

（1）当取指令和数据存取发生竞争时

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

* 存储体上溢时，状态数为 19。

5.1.7 浮点指令

表 5.8 浮点指令

指令		操作码	运行	执行状态	T 位	兼容性		
						SH2E	SH4	新 SH-2A/ SH2A-FPU
FABS	FRn	1111nnnn01011101	|FRn|→FRn	1	—	○	○	
FABS	DRn	1111nnn001011101	|DRn|→DRn	1	—		○	
FADD	FRm,FRn	1111nnnnmmmm0000	FRn+FRm→FRn	1	—	○	○	
FADD	DRm,DRn	1111nnn0mmm00000	DRn+DRm→DRn	6	—		○	
FCMP/EQ	FRm,FRn	1111nnnnmmmm0100	(FRn=FRm)? 1:0→T	1	比较结果	○	○	
FCMP/EQ	DRm,DRn	1111nnn0mmm00100	(DRn=DRm)? 1:0→T	2	比较结果		○	
FCMP/GT	FRm,FRn	1111nnnnmmmm0101	(FRn>FRm)? 1:0→T	1	比较结果	○	○	
FCMP/GT	DRm,DRn	1111nnn0mmm00101	(DRn>DRm)? 1:0→T	2	比较结果		○	
FCNVDS	DRm,FPUL	1111mmm010111101	(float)DRm→FPUL	2	—		○	
FCNVSD	FPUL,DRn	1111nnn010101101	(double)FPUL→DRn	2	—		○	
FDIV	FRm,FRn	1111nnnnmmmm0011	FRn/FRm→FRn	10	—	○	○	
FDIV	DRm,DRn	1111nnn0mmm00011	DRn/DRm→DRn	23	—		○	
FLDI0	FRn	1111nnnn10001101	0×00000000→FRn	1	—	○	○	
FLDI1	FRn	1111nnnn10011101	0×3F800000→FRn	1	—	○	○	
FLDS	FRm,FPUL	1111mmmm00011101	FRm→FPUL	1	—	○	○	
FLOAT	FPUL,FRn	1111nnnn00101101	(float)FPUL→FRn	1	—	○	○	
FLOAT	FPUL,DRn	1111nnn000101101	(double)FPUL→DRn	2	—		○	
FMAC	FR0,FRm,FRn	1111nnnnmmmm1110	FR0×FRm+FRn→FRn	1	—	○	○	
FMOV	FRm,FRn	1111nnnnmmmm1100	FRm→FRn	1	—	○	○	
FMOV	DRm,DRn	1111nnn0mmm01100	DRm→DRn	2	—		○	
FMOV.S	@(R0,Rm), FRn	1111nnnnmmmm0110	(R0+Rm)→FRn	1	—	○	○	
FMOV.D	@(R0,Rm),DRn	1111nnn0mmmm0110	(R0+Rm)→DRn	2	—		○	
FMOV.S	@Rm+,FRn	1111nnnnmmmm1001	(Rm)→FRn,Rm+=4	1	—	○	○	
FMOV.D	@Rm+,DRn	1111nnn0mmmm1001	(Rm)→DRn,Rm+=8	2	—		○	
FMOV.S	@Rm,FRn	1111nnnnmmmm1000	(Rm)→FRn	1	—	○	○	
FMOV.D	@Rm,DRn	1111nnn0mmmm1000	(Rm)→DRn	2	—		○	
FMOV.S	@(disp12,Rm),FRn	0011nnnnmmmm0001 0111dddddddddddd	(disp×4+Rm)→FRn	1	—			○
FMOV.D	@(disp12,Rm),DRn	0011nnn0mmmm0001 0111dddddddddddd	(disp×8+Rm)→DRn	2	—			○
FMOV.S	FRm,@(R0,Rn)	1111nnnnmmmm0111	FRm→(R0+Rn)	1	—	○	○	
FMOV.D	DRm, @(R0,Rn)	1111nnnnmmm00111	DRm→(R0+Rn)	2	—		○	
FMOV.S	FRm,@-Rn	1111nnnnmmmm1011	Rn-=4,FRm→(Rn)	1	—	○	○	
FMOV.D	DRm,@-Rn	1111nnnnmmm01011	Rn-=8,DRm→(Rn)	2	—		○	
FMOV.S	FRm,@Rn	1111nnnnmmmm1010	FRm→(Rn)	1	—	○	○	
FMOV.D	DRm,@Rn	1111nnnnmmm01010	DRm→(Rn)	2	—		○	

指令		操作码	运行	执行状态	T 位	兼容性		
						SH2E	SH4	新 SH-2A/ SH2A-FPU
FMOV.S	FRm,@(disp12,Rn)	0011nnnnmmmm0001 0011dddddddddddd	FRm→(disp×4+Rn)	1	—			○
FMOV.D	DRm,@(disp12,Rn)	0011nnnnmmm00001 0011dddddddddddd	DRm→(disp×8+Rn)	2	—			○
FMUL	FRm,FRn	1111nnnnmmmm0010	FRn×FRm→FRn	1	—	○	○	
FMUL	DRm,DRn	1111nnn0mmm00010	DRn×DRm→DRn	6	—		○	
FNEG	FRn	1111nnnn01001101	-FRn→FRn	1	—	○	○	
FNEG	DRn	1111nnn001001101	-DRn→DRn	1	—		○	
FSCHG		1111001111111101	FPSCR.SZ=~FPSCR.SZ	1	—		○	
FSQRT	FRn	1111nnnn01101101	$\sqrt{FRn}$→FRn	9	—		○	
FSQRT	DRn	1111nnn001101101	$\sqrt{DRn}$→DRn	22	—		○	
FSTS	FPUL,FRn	1111nnnn00001101	FPUL→FRn	1	—	○	○	
FSUB	FRm,FRn	1111nnnnmmmm0001	FRn-FRm→FRn	1	—	○	○	
FSUB	DRm,DRn	1111nnn0mmm00001	DRn-DRm→DRn	6	—		○	
FTRC	FRm,FPUL	1111mmmm00111101	(long)FRm→FPUL	1	—	○	○	
FTRC	DRm,FPUL	1111mmm000111101	(long)DRm→FPUL	2	—		○	

5.1.8 有关 FPU 的 CPU 指令

表 5.9 有关 FPU 的 CPU 指令

指令		操作码	运行	执行状态	T 位	兼容性		
						SH2E	SH4	新 SH-2A/ SH2A-FPU
LDS	Rm,FPSCR	0100mmmm01101010	Rm→FPSCR	1	—	○	○	
LDS	Rm,FPUL	0100mmmm01011010	Rm→FPUL	1	—	○	○	
LDS.L	@Rm+,FPSCR	0100mmmm01100110	(Rm)→FPSCR,Rm+=4	1	—	○	○	
LDS.L	@Rm+,FPUL	0100mmmm01010110	(Rm)→FPUL,Rm+=4	1	—	○	○	
STS	FPSCR,Rn	0000nnnn01101010	FPSCR→Rn	1	—	○	○	
STS	FPUL,Rn	0000nnnn01011010	FPUL→Rn	1	—	○	○	
STS.L	FPSCR,@-Rn	0100nnnn01100010	Rn-=4,FPCSR→(Rn)	1	—	○	○	
STS.L	FPUL,@-Rn	0100nnnn01010010	Rn-=4,FPUL→(Rn)	1	—	○	○	

5.1.9 位操作指令

表 5.10 位操作指令

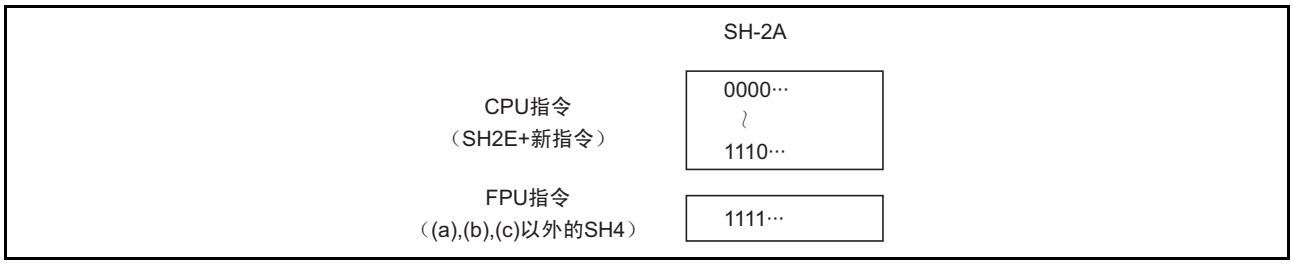
指令	操作码	运行	执行状态	T 位	兼容性		
					SH2E	SH4	新 SH-2A/ SH2A-FPU
BAND.B #imm3,@(disp12,Rn)	0011nnnn0iii1001 0100dddddddddddd	(imm of(disp+ Rn))&T → T	3	运算结果			○
BANDNOT.B #imm3,@(disp12,Rn)	0011nnnn0iii1001 1100dddddddddddd	~(imm of(disp+ Rn))&T → T	3	运算结果			○
BCLR.B #imm3,@(disp12,Rn)	0011nnnn0iii1001 0000dddddddddddd	0→ (imm of (disp+ Rn))	3	—			○
BCLR #imm3,Rn	10000110nnnn0iii	0→ imm of Rn	1	—			○
BLD.B #imm3,@(disp12,Rn)	0011nnnn0iii1001 0011dddddddddddd	(imm of (disp+Rn)) → T	3	运算结果			○
BLD #imm3,Rn	10000111nnnn1iii	imm of Rn →T	1	运算结果			○
BLDNOT.B #imm3,@(disp12,Rn)	0011nnnn0iii1001 1011dddddddddddd	~(imm of (disp+Rn)) → T	3	运算结果			○
BOR.B #imm3,@(disp12,Rn)	0011nnnn0iii1001 0101dddddddddddd	(imm of (disp+ Rn)) | T → T	3	运算结果			○
BORNOT.B #imm3,@(disp12,Rn)	0011nnnn0iii1001 1101dddddddddddd	~(imm of (disp+ Rn)) | T → T	3	运算结果			○
BSET.B #imm3,@(disp12,Rn)	0011nnnn0iii1001 0001dddddddddddd	1→(imm of (disp+Rn))	3	—			○
BSET #imm3,Rn	10000110nnnn1iii	1→ imm of Rn	1	—			○
BST.B #imm3,@(disp12,Rn)	0011nnnn0iii1001 0010dddddddddddd	T→(imm of (disp+Rn))	3	—			○
BST #imm3,Rn	10000111nnnn0iii	T→ imm of Rn	1	—			○
BXOR.B #imm3,@(disp12,Rn)	0011nnnn0iii1001 0110dddddddddddd	(imm of (disp+ Rn)) ^ T → T	3	运算结果			○

第 6 章 各指令的说明

6.1 新指令的概要

在 SH-2A/SH2A-FPU 中，除分配给 SH2E 的 CPU 指令 （最高 4 位为 0000 ～ 1110 的操作码）与 SH4 的 FPU 指令 （最高 4 位为 1111 的操作码）的操作码以外，在空白区域追加有新指令。但是，不支持 SH4 的 FPU 指令中：(a) 指定 XDm/XDn 的 FMOV 指令、 (b)FRCHG 指令、 (c)FIPR 与 FTRV 指令。

本章对各个新指令进行详细说明。



新指令为以下 (1) ～ (14)。 (1) ～ (3) 为 32 位固定长指令、 (4) ～ (14) 为 16 位固定长指令。

(1) 立即传送指令

MOVI20、 MOVI20S

是将操作码内的 20 位立即数传送到寄存器的指令。因为本指令组合可以容易的生成 28 位地址，因此内部存储器的地址最大可指定为 256M 字节。

(2) 结构体存取用指令

MOV.B/W/L Rm, @(disp12, Rn) , MOV.B/W/L @(disp12, Rm) , Rn

MOVU.B/W @(disp12, Rm) , Rn

FMOV.S FRm, @(disp12, Rn) , FMOV.S @(disp12, Rm) , FRn

FMOV.D DRm, @(disp12, Rn) , FMOV.D @(disp12, Rm) , DRn

是指定配置在操作码中的 12 位的位移量后参照存储器的指令。另外，追加自动执行零扩展的无符号加载指令 MOVU。

(3) 位操作指令 （存储器对象）

BAND.B #imm3, @(disp12, Rn) , BOR.B #imm3, @(disp12, Rn)

BCLR.B #imm3, @(disp12, Rn) , BSET.B #imm3, @(disp12, Rn)

BST.B #imm3, @(disp12, Rn) , BLD.B #imm3, @(disp12, Rn)

BXOR.B #imm3, @(disp12, Rn)

BANDNOT.B #imm3, @(disp12, Rn) , BORNOT.B #imm3, @(disp12, Rn)

BLDNOT.B #imm3, @(disp12, Rn)

BAND.B 指令、 BOR.B 指令与 BXOR.B 指令对存储器中的 1 位与 T 位进行逻辑运算，并将其结果保存到 T 位的指令。 BCLR.B 指令与 BSET.B 指令是操作存储器中的 1 位的指令。 BST.B 指令与 BLD.B 指令是执行存储器中的 1 位与 T 位间传送的指令。

BANDNOT.B 指令与 BORNOT.B 指令，是对存储器中的 1 位取反后的值与 T 位进行逻辑运算，并将其结果保存到 T 位的指令。

BLDNOT.B 指令是取反存储器中的 1 位并保存到 T 位的指令。

指定以外的位不受影响。

(4) 位操作指令 （通用寄存器对象）

BCLR #imm3, Rn, BSET #imm3, Rn

BST #imm3, Rn, BLD #imm3, Rn

BCLR 指令与 BSET 指令是操作通用寄存器 Rn 的 LSB 侧 8 位中的 1 位的指令。 BST 指令与 BLD 指令是执行通用寄存器 Rn 的 LSB 侧 8 位中 1 位与 T 位间的传送的指令。指定以外的位不受影响。

(5) 乘法结果 Rn 存储指令

MULR

是进行 32 位 ×32 位的乘法运算，并将其结果的低 32 位存储到通用寄存器 Rn 的指令。

(6) 批量除法指令

DIVS,DIVU

批量进行 32 位 ÷32 位除法运算的指令。 DIVU 指令是将无符号的数据进行除法运算的指令， DIVS 指令是将带符号的数据进行除法运算的指令。

(7) 饱和值比较指令

CLIPS,CLIPU

与饱和值进行比较，通用寄存器 Rn 在高于饱和上限值及低于饱和下限值时，分别将饱和上限值与饱和下限值保存到通用寄存器 Rn。另外，只支持字节和字的饱和值。

(8) 桶式移位指令

SHAD,SHLD

是移动任意位的指令。具备算术移位与逻辑移位 2 种指令。

(9) 多个寄存器保存 / 返回指令

MOVML,MOVMU

是将多个连续的寄存器保存到存储器或从存储器返回的指令。可在指定通用寄存器 Rn 后，保存及返回高位或低位 （与指定 Rn 相比）连续的通用寄存器。

(10) T 位取反传送指令

MOVRT,NOTT

取反 T 位并将结果传送到通用寄存器 Rn 或者 T 位的指令。

(11) 寄存器存储体关联指令

RESBANK,STBANK,LDBANK

是为高速化中断处理而装载的有关寄存器存储体的指令。

(12) 反向堆栈传送指令

MOV.B/W/L

是堆栈延长方向相反的传送指令。

(13) 无延迟槽的无条件转移指令

JSR/N,RTS/N

为了通过减少无用的 NOP 指令削减代码长度，提供无延迟槽的指令。

(14) 高速缓存关联指令

PREF

提供 SH3-DSP 的高速缓存关联指令。

6.2 指令说明的格式

按照以下格式说明本章的阅读方法。

指令名称　　指令功能 （英文）　　指令分类

指令功能　　指令系统兼容

格式	操作概略	操作码	执行状态	T 位
用汇编程序的输入格式表示。 Imm,disp 为数值、表达式或者符号。	表示操作概略。	按照 MSB←→LSB 的顺序表示。	是无等待时的值。	表示执行指令后的 T 位的值。

(1) 说明

对运行进行说明。

(2) 注意

说明在使用指令时需要特别注意的事项。

(3) 操作内容

用 C 表示操作内容。作为理解运行的参考加以记述。

假定使用以下资源。

```
unsigned  char          Read_Byte (unsigned  long  Addr) ;
unsigned  short         Read_Word (unsigned  long  Addr) ;
unsigned  int           Read_Int (unsigned  long  Addr) ;
unsigned  long          Read_Long (unsigned  long  Addr) ;
unsigned  double        Read_Quad (unsigned  long  Addr);
```

返回各种长度地址 Addr 的内容。如果不从地址 2n 开始读取字数据或者不从地址 4n 开始读取长字数据，就会检测为地址错误。

```
unsigned  long      Read_Bank_Long (unsigned  long  Addr) ;
```

返回地址 Addr 的内容所指向的寄存器存储体内入口的内容。

```
unsigned  char      Write_Byte (unsigned long Addr, unsigned long Data) ;
unsigned  short     Write_Word (unsigned long Addr, unsigned long Data) ;
unsigned  int       Write_Int(unsigned long Addr, unsigned long Data) ;
unsigned  long      Write_Long (unsigned long Addr, unsigned long Data) ;
unsigned  double    Write_Quad (unsigned long Addr, unsigned long Data);
```

以各种长度将数据 Data 写入地址 Addr。如果不从 2n 地址开始写入字数据或者不从地址 4n 开始写入长字数据，就会检测为地址错误。

```
unsigned  long  Write_Bank_Long (unsigned  long Add、unsigned long Data);
```

将数据 Data 写入地址 Addr 的内容所指示的寄存器存储体内的入口。

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

各寄存器的本体

```
struct  BANK {
            unsigned  long  Rn_BANK[15];
            unsigned  long  GBR_BANK;
            unsigned  long  MACH_BANK;
            unsigned  long  MACL_BANK;
            unsigned  long  PR_BANK;
            unsigned  long  IVN;
};
BANK  Register_Bank[512];
```

寄存器存储体的结构定义

（VTO：中断向量表地址偏移）

```
struct SR0 {
            unsigned  long  dummy0:17 ;
            unsigned  long  BO0:1
            unsigned  long  CS0:1;
            unsigned  long  dummy1:3;
            unsigned  long  M0:1 ;
            unsigned  long  Q0:1 ;
            unsigned  long  I0:4 ;
            unsigned  long  dummy2:2;
            unsigned  long  S0:1;
            unsigned  long  T0:1;
} ;
```

SR 的结构定义

```
#define BO ( ( * (struct SR0 * ) ( &SR) ).BO0)
#define CS ( ( * (struct SR0 * ) ( &SR) ).CS0)
#define M ( ( * (struct SR0 * ) ( &SR) ).M0)
#define Q ( ( * (struct SR0 * ) ( &SR) ).Q0)
#define I ( ( * (struct SR0 * ) ( &SR) ).I0)
#define S ( ( * (struct SR0 * ) ( &SR) ).S0)
#define T ( ( * (struct SR0 * ) ( &SR) ).T0)
```

SR 内的位定义

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

错误显示函数

浮点的定义说明如下

```
#define PZERO               0
#define NZERO               1
#define DENORM              2
#define NORM                3
#define PINF                4
#define NINF                5
#define qNaN                6
#define sNaN                7
#define EQ                  0
#define GT                  1
#define LT                  2
#define UO                  3
#define INVALID             4
#define FADD                0
#define FSUB                1

#define CAUSE               0x0003f000/* FPSCR(bit17-12) */
#define SET_E               0x00020000/* FPSCR(bit17) */
#define SET_V               0x00010040/* FPSCR(bit16,6) */
#define SET_Z               0x00008020/* FPSCR(bit15,5) */
#define SET_O               0x00004010/* FPSCR(bit14,4) */
#define SET_U               0x00002008/* FPSCR(bit13,3) */
#define SET_I               0x00001004/* FPSCR(bit12,2) */
#define ENABLE_VOUI         0x00000b80/* FPSCR(bit11,9-7) */
#define ENABLE_V            0x00000800/* FPSCR(bit11) */
#define ENABLE_Z            0x00000400/* FPSCR(bit10) */
#define ENABLE_OUI          0x00000380/* FPSCR(bit9-7) */
#define ENABLE_I            0x00000080/* FPSCR(bit7) */
#define FLAG                0x0000007C/* FPSCR(bit6-2) */

#define FPSCR_FR            FPSCR>>21&1
```

```
#define FPSCR_PR            FPSCR>>19&1
#define FPSCR_DN            FPSCR>>18&1
#define FPSCR_I             FPSCR>>12&1
#define FPSCR_RM            FPSCR&1
#define FR_HEX              frf.l[ FPSCR_FR]
#define FR                  frf.f[ FPSCR_FR]
#define DR_HEX              frf.l[ FPSCR_FR]
#define DR                  frf.d[ FPSCR_FR]

union {
             int  l[2][16];
             float  f[2][16];
             double d[2][8];
} frf;
int FPSCR;

int sign_of(int n)
{
             return(FR_HEX[n]>>31);
}
int data_type_of(int n) {
int abs;
             abs = FR_HEX[n] & 0x7fffffff;
             if(FPSCR_PR == 0) {/* 单精度 */
                if(abs < 0x00800000){
                   if((FPSCR_DN == 1) || (abs == 0x00000000)){
                      if(sign_of(n) == 0){zero(n, 0); return(PZERO);}
                      else               {zero(n, 1); return(NZERO);}
                   }
                   else                          return(DENORM);
             }
             else if(abs < 0x7f800000)return(NORM);
             else if(abs == 0x7f800000) {
                   if(sign_of(n) == 0)           return(PINF);
                   else                          return(NINF);
                   }
                   else if(abs < 0x7fc00000)     return(qNaN);
                   else                          return(sNaN);
}
else {/* 双精度 */
  if(abs < 0x00100000){
      if((FPSCR_DN == 1) || ((abs == 0x00000000) && (FR_HEX[n+1] == 0x00000000){
          if(sign_of(n) == 0){zero(n, 0); return(PZERO);}
          else    {zero(n, 1); return(NZERO);}
  }
  else            return(DENORM);
```

```
        }
    }
        else if(abs < 0x7ff00000)                return(NORM);
        else if((abs == 0x7ff00000) &&
              (FR_HEX[n+1] == 0x00000000)) {
           if(sign_of(n) == 0)return(PINF);
           else               return(NINF);
        }
              else if(abs < 0x7ff80000)return(qNaN);
              else                     return(sNaN);
              }
        }
        void register_copy(int m,n)
        {
                                        FR[n]   = FR[m];
                  if(FPSCR_PR == 1)     FR[n+1] = FR[m+1];
        }
        void normal_faddsub(int m,n,type)
        {
        union {
                  float f;
                  int l;
        }         dstf,srcf;
        union {
                  double d;
                  int l[2];
        }         dstd,srcd;
        union {                         /*  “long double” 的格式 :*/
                  long double x;        /*1-bit 符号 */
                  int l[4];             /*15-bit 指数 */
        }         dstx;                 /*112-bit 小数 */
                  if(FPSCR_PR == 0) {
                     if(type == FADD)          srcf.f =  FR[m];
                     else                     srcf.f = -FR[m];
                     dstd.d = FR[n]; /* 从单精度转换为双精度 */
                     dstd.d += srcf.f;
                     if(((dstd.d == FR[n]) && (srcf.f != 0.0)) ||
                        ((dstd.d == srcf.f) && (FR[n] != 0.0))) {
                        set_I( );
                        if(sign_of(m)^ sign_of(n)) {
                            dstd.l[1] -= 1;
                        if(dstd.l[1] == 0xffffffff) dstd.l[0] -= 1;
                        }
                     }
                     if(dstd.l[1] & 0x1fffffff) set_I( );
                     dstf.f += srcf.f; /* 就近舍入 */
```

```
                if(FPSCR_RM == 1) {
                   dstd.l[1] &= 0xe0000000; /* 向 0 舍入 */
                   dstf.f = dstd.d;
                }
                check_single_exception(&FR[n],dstf.f);
             } else {
                if(type == FADD)         srcd.d =  DR[m>>1];
                else                     srcd.d = -DR[m>>1];
                dstx.x = DR[n>>1]; /* 从双精度转换为扩展双精度 */
                dstx.x += srcd.d;
                if(((dstx.x == DR[n>>1]) && (srcd.d != 0.0)) ||
                   ((dstx.x == srcd.d) && (DR[n>>1] != 0.0)) ) {
                     set_I( );
                     if(sign_of(m)^ sign_of(n)) {
                       dstx.l[3] -= 1;
                       if(dstx.l[3] == 0xffffffff) {dstx.l[2] -= 1;
                       if(dstx.l[2] == 0xffffffff) {dstx.l[1] -= 1;
                       if(dstx.l[1] == 0xffffffff) {dstx.l[0] -= 1;}}}
                   }
             }
             if((dstx.l[2] & 0x0fffffff) || dstx.l[3]) set_I( );
             dst.d += srcd.d; /* 就近舍入 */
             if(FPSCR_RM == 1) {
                dstx.l[2] &= 0xf0000000; /* 向 0 舍入 */
                dstx.l[3]  = 0x00000000;
                dst.d = dstx.x;
             }
             check_double_exception(&DR[n>>1] ,dst.d);
      }
}
void normal_fmul(int m,n)
{
union {
      float f;
           int l;
}     tmpf;
union {
      double d;
      int l[2];
}     tmpd;
union {
      long double x;
      int l[4];
}     tmpx;
      if(FPSCR_PR == 0) {
           tmpd.d = FR[n]; /* 从单精度到双精度 */
```

```
            tmpd.d *= FR[m]; /* 正确生成 */
            tmpf.f *= FR[m]; /* 就近舍入 */
            if(tmpf.f != tmpd.d) set_I( );
            if((tmpf.f > tmpd.d) && (FPSCR_RM == 1)) {
                                    tmpf.l -= 1; /* 向 0 舍入 */
            }
            check_single_exception(&FR[n],tmpf.f);
      } else {
            tmpx.x = DR[n>>1]; /* 从单精度到双精度 */
            tmpx.x *= DR[m>>1]; /* 正确生成 */
            tmpd.d *= DR[m>>1]; /* 就近舍入 */
            if(tmpd.d != tmpx.x) set_I( );
            if(tmpd.d > tmpx.x) && (FPSCR_RM == 1)) {
               tmpd.l[1] -= 1; /* 向 0 舍入 */
            if(tmpd.l[1] == 0xffffffff) tmpd.l[0] -= 1;
            }
      check_double_exception(&DR[n>>1], tmpd.d);
      }
}
void check_single_exception(float *dst,result)
{
union {
      float f;
      int l;
}     tmp;
float abs;
      if(result < 0.0)tmp.l = 0xff800000; /* - 无穷大 */
      else            tmp.l = 0x7f800000; /* + 无穷大 */
      if(result == tmp.f) {
                  set_O( ); set_I( );
                  if(FPSCR_RM == 1){
                              tmp.l -= 1; /* 规格化数的最大值 */
                              result = tmp.f;
                                    }
      }
      if(result < 0.0) abs = -result;
      else              abs =  result;
      tmp.l = 0x00800000; /* 规格化数的最小值 */
      if(abs < tmp.f) {
            if((FPSCR_DN == 1) && (abs != 0.0)) {
               set_I( );
               if(result < 0.0)         result = -0.0; /* 使非规格化数为零。 */
               else                       result =  0.0;
      }
      if(FPSCR_I == 1) set_U( );
}
```

```
      if(FPSCR & ENABLE_OUI) fpu_exception_trap( );
      else                        *dst = result;
}
      void check_double_exception(double *dst,result)
      {
      union {
            double d;
            int l[2];
      }     tmp;
      double abs;
            if(result < 0.0)tmp.l[0] = 0xfff00000; /* - 无穷大 */
            else            tmp.l[0] = 0x7ff00000; /* + 无穷大 */
                            tmp.l[1] = 0x00000000;
            if(result == tmp.d)
               set_O( ); set_I( );
               if(FPSCR_RM == 1) {
                   tmp.l[0] -= 1;
                   tmp.l[1] = 0xffffffff;
                   result = tmp.d; /* 规格化数的最大值 */
      }
}
            if(result < 0.0)abs = -result;
            else                    abs =  result;
            tmp.l[0] = 0x00100000; /* 规格化数的最小值 */
            tmp.l[1] = 0x00000000;
            if(abs < tmp.d) {
            if((FPSCR_DN == 1) && (abs != 0.0)) {
                 set_I( );
                 if(result < 0.0)        result = -0.0; /* 使非规格化数为零。 */
                    else                 result =  0.0;
      }
      if(FPSCR_I == 1) set_U( );
}
  if(FPSCR & ENABLE_OUI)fpu_exception_trap( );
                 else                    *dst = result;
      }
      int check_product_invalid(int m,n)
      {
            return(check_product_infinity(m,n)  &&
                 ((data_type_of(m) == PZERO) || (data_type_of(n) == PZERO) ||
                 (data_type_of(m) == NZERO) || (data_type_of(n) == NZERO)));
}
int check_product_infinity(int m,n)
{
            return((data_type_of(m) == PINF) || (data_type_of(n) == PINF) ||
                   (data_type_of(m) == NINF) || (data_type_of(n) == NINF));
```

```
}
int check_positive_infinity(int m,n)
{
  return(((check_product_infinity(m,n) && (~sign_of(m)^ sign_of(n))) ||
      ((check_product_infinity(m+1,n+1) && (~sign_of(m+1)^ sign_of(n+1))) ||
      ((check_product_infinity(m+2,n+2) && (~sign_of(m+2)^ sign_of(n+2))) ||
      ((check_product_infinity(m+3,n+3) && (~sign_of(m+3)^ sign_of(n+3))));
}
int check_negative_infinity(int m,n)
{
  return(((check_product_infinity(m,n) && (sign_of(m)^ sign_of(n))) ||
      ((check_product_infinity(m+1,n+1) && (sign_of(m+1)^ sign_of(n+1))) ||
      ((check_product_infinity(m+2,n+2) && (sign_of(m+2)^ sign_of(n+2))) ||
      ((check_product_infinity(m+3,n+3) && (sign_of(m+3)^ sign_of(n+3))));
}
void clear_cause ( ) {FPSCR &= ~CAUSE;}
void set_E( ) {FPSCR |= SET_E; fpu_exception_trap( );}
void set_V( ) {FPSCR |= SET_V;}
void set_Z( ) {FPSCR |= SET_Z;}
void set_O( ) {FPSCR |= SET_O;}
void set_U( ) {FPSCR |= SET_U;}
void set_I( ) {FPSCR |= SET_I;}
void invalid(int n)
{
      set_V( );
      if((FPSCR & ENABLE_V) == 0 qnan(n);
      else        fpu_exception_trap( );
}

void dz(int n,sign)
{
      set_Z( );
      if((FPSCR & ENABLE_Z) == 0 inf(n,sign);
      else        fpu_exception_trap( );
}
void zero(int n,sign)
{
      if(sign == 0)         FR_HEX [n]   = 0x00000000;
      else                  FR_HEX [n]   = 0x80000000;
      if (FPSCR_PR==1)      FR_HEX [n+1] = 0x00000000;
}
void inf(int n,sign) {
      if (FPSCR_PR==0) {
      if(sign == 0          FR_HEX [n]   = 0x7f800000;
      else                  FR_HEX [n]   = 0xff800000;
      } else {
```

```
        if(sign == 0          FR_HEX [n]   = 0x7ff00000;
        else                  FR_HEX [n]   = 0xfff00000;
                              FR_HEX [n+1] = 0x00000000;
        }
}
void qnan(int n)
{
        if (FPSCR_PR==0       FR[n]   = 0x7fbfffff;
        else {                FR[n]   = 0x7ff7ffff;
                              FR[n+1] = 0xffffffff;
        }
}
```

(4) 使用例

用汇编程序助记符示例，表示指令执行前后的状态。

斜体字 （例：*.align*）表示汇编控制指令。汇编控制指令的含义如下所示，详细内容请参照 《交叉汇编程序的用户手册》。

.org	设定定位计数器
.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 字节的边界
.align 4	调整 4 字节的边界
.align 32	调整 32 字节的边界
.arepeat 16	重复展开 16 次
.arepeat 32	重复展开 32 次
.aendr	结束指定次数的重复展开

【注】 SH 系列的交叉汇编程序 Ver 1.0 不支持带条件的汇编功能。

6.3 新指令

6.3.1 BAND Bit AND 位操作指令

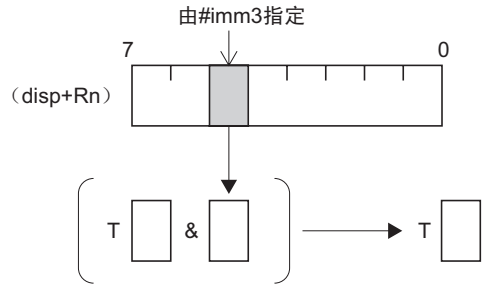
位与 SH-2A/SH2A-FPU （新）

格式	操作概略	操作码	执行状态	T 位
BAND.B #imm3,@(disp12,Rn)	(<imm>of （disp+Rn）)&T→T	0011nnnn0iii1001 0100dddddddddddd	3	运算结果

(1) 说明

取 (disp+Rn) 所示地址的存储器中指定的 1 位与 T 位的逻辑与，并将其结果保存到 T 位。位号码由 3 位的立即数指定。在本指令下以字节为单位从存储器读取数据。

```
BAND.B #imm3, @(disp12, Rn)
```



(2) 操作内容

```
BANDM (long d, long i, long n)  /*BAND.B #imm3, @(disp12, Rn) */
  {
    long disp, imm, temp, assignbit;

      disp = (0x00000FFF & (long)d );
      imm= (0x00000007&(long)i);
      temp= (long) Read_Byte ( R[n]+disp);
      assignbit =(0x00000001<<imm)&temp;
      if((T==0)||(assignbit==0)) T=0;
      else T=1;
      PC+=4;
}
```

(3) 使用例

```
BAND.B#H'5, @(2, R0)                ;执行前  @(R0+2)=H'DF, T=1
                                    ;执行后  @(R0+2)=H'DF, T=0
```

6.3.2 BANDNOT Bit ANDNOT 位操作指令

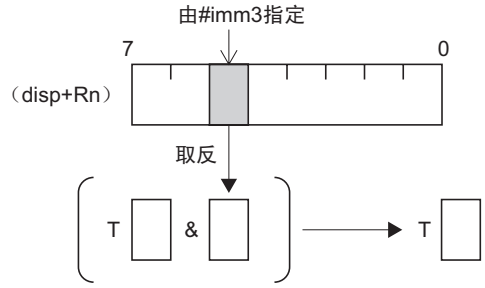
位非与 SH-2A/SH2A-FPU （新）

格式	操作概略	操作码	执行状态	T 位
BANDNOT.B #imm3,@(disp12,Rn)	~(<imm>of （disp+Rn）)&T→T	0011nnnn0iii1001 1100dddddddddddd	3	运算结果

(1) 说明

取 (disp+Rn) 所示地址的存储器中指定的 1 位取反后的值与 T 位的逻辑与，并将其结果保存到 T 位。位号码由 3 位的立即数指定。在本指令下以字节为单位从存储器读取数据。

```
BANDNOT.B #imm3, @(disp12, Rn)
```



(2) 操作内容

```
BANDNOTM (long d, long i, long n)  /*BANDNOT.B #imm3, @(disp12, Rn) */
 {
  long disp, imm, temp, assignbit;

    disp = (0x00000FFF & (long)d );
    imm= (0x00000007&(long)i);
    temp= (long) Read_Byte ( R[n]+disp);
    assignbit =(0x00000001<<imm)&temp;
    if((T==1)&&(assignbit==0)) T=1;
    else T=0;
    PC+=4;
}
```

(3) 使用例

```
BANDNOT.B#H'5, @(2, R0)              ;执行前  @(R0+2)=H'20, T=1
                                     ;执行后  @(R0+2)=H'20, T=0
```

6.3.3 BCLR Bit CLeaR 位操作指令

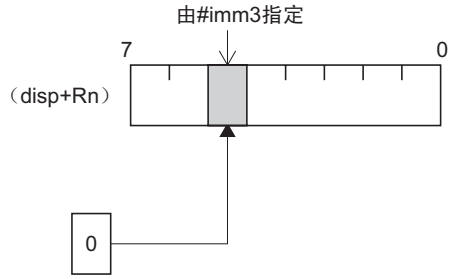
位清除 SH-2A/SH2A-FPU （新）

格式	操作概略	操作码	执行状态	T 位
BCLR.B #imm3,@(disp12,Rn)	0→(<imm>of （disp+Rn）)	0011nnnn0iii1001 0000dddddddddddd	3	—
BCLR #imm3, Rn	0→ <imm>of Rn	10000110nnnn0iii	1	—

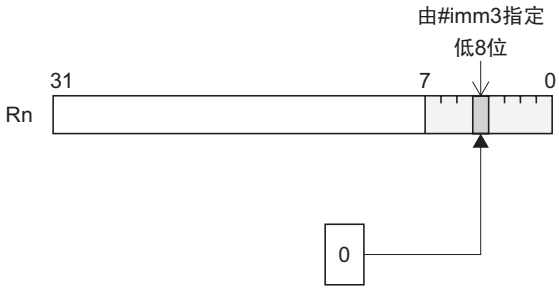
(1) 说明

清除 (disp+Rn) 所示地址的存储器或者通用寄存器 Rn 的 LSB 侧 8 位中指定的 1 位。位号码由 3 位的立即数指定。在 BCLR.B 指令下，以字节为单位从存储器读取数据后，执行指定位的清除，并将执行后的数据以字节为单位写入存储器。

```
BCLR.B #imm3, @(disp12, Rn)
```




```
BCLR #imm3, Rn
```



(2) 操作内容

```
BCLRM (long d, long i, long n)  /*BCLR.B #imm3, @(disp12, Rn) */
{
  long disp, imm, temp;

     disp = (0x00000FFF & (long)d );
     imm= (0x00000007&(long)i);
     temp= (long) Read_Byte ( R[n]+disp);
     temp&=(~(0x00000001<<imm));
     Write_Byte (R[n]+disp, temp);
     PC+=4;
}

BCLR (long i, long n)  /*BCLR #imm3, Rn */
 {
   long imm, temp;

   imm= (0x00000007 &(long)i);
   R[n]&=( . (0x00000001<<imm))
  PC+=2;
  }
```

(3) 使用例

```
BCLR.B#H'5, @(2, R0)                ;执行前  @(R0+2)=H'FF
                                    ;执行后  @(R0+2)=H'DF
BCLR#H'4, R0                        ;执行前  R0=H'FFFFFFFF
                                    ;执行后  R0=H'FFFFFFFF
```

6.3.4 BLD Bit LoaD 位操作指令

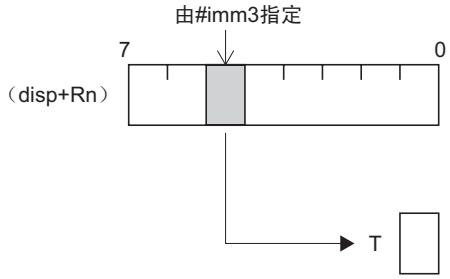
位加载 SH-2A/SH2A-FPU （新）

格式	操作概略	操作码	执行状态	T 位
BLD.B #imm3,@(disp12,Rn)	(<imm>of （disp+Rn）) →T	0011nnnn0iii1001 0011dddddddddddd	3	运算结果
BLD #imm3, Rn	<imm>of Rn→ T	10000111nnnn1iii	1	运算结果

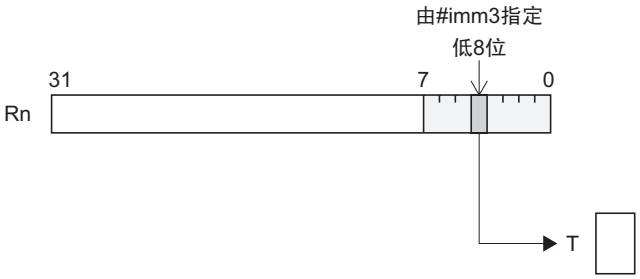
(1) 说明

将 (disp+Rn) 所示地址的存储器或者通用寄存器 Rn 的 LSB 侧 8 位中指定的 1 位保存到 T 位。位号码由 3 位的立即数指定。由 BLD.B 指令下，以字节为单位从存储器读取数据。

```
BLD.B #imm3, @(disp12, Rn)
```




```
BLD #imm3, Rn
```



(2) 操作内容

```
BLDM (long d, long i, long n)  /*BLD.B #imm3, @(disp12, Rn) */
 {
   long disp, imm, temp,assignbit;

  disp = (0x00000FFF & (long)d );
  imm= (0x00000007&(long)i);
  temp = (long) Read_Byte ( R[n]+disp);
  assignbit=(0x00000001<<imm)&temp;
  if(assignbit==0) T=0;
  else T=1;
  PC+=4;
}
BLD (long i, long n)  /*BLD #imm3, Rn */
 {
   long imm, assignbit;

   imm= (0x00000007&(long)i);
   assignbit=(0x00000001<<imm)&R[n];
   if(assignbit ==0) T=0;
   else T=1;
   PC+=2;
  }
```

(3) 使用前

```
BLD.B#H'5, @(2, R0)                    ;执行前  @(R0+2)=H'20, T=0
                                       ;执行后  @(R0+2)=H'20, T=1
BLD#H'4, R0                            ;执行前  R0=H'000000EF, T=1
                                       ;执行后  R0=H'000000EF, T=0
```

6.3.5 BLDNOT Bit LoaD NOT 位操作指令

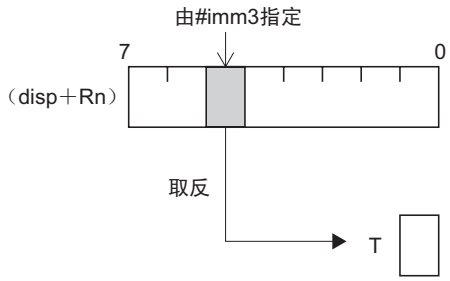
位非加载 SH-2A/SH2A-FPU （新）

格式	操作概略	操作码	执行状态	T 位
BLDNOT.B #imm3,@(disp12,Rn)	~(<imm>of （disp+Rn）) →T	0011nnnn0iii1001 1011dddddddddddd	3	运算结果

(1) 说明

将 (disp+Rn) 所示地址的存储器中指定的 1 位取反，并将结果保存到 T 位。位号码由 3 位的立即数指定。在 BLDNOT.B 指令下，以字节为单位从存储器读取数据。

```
BLDNOT.B #imm3, @(disp12, Rn)
```



(2) 操作内容

```
BLDNOTM (long d, long i, long n)  /*BLDNOT.B #imm3, @(disp12, Rn) */
 {
   long disp, imm, temp,assignbit;

  disp = (0x00000FFF & (long)d );
  imm= (0x00000007&(long)i);
  temp = (long) Read_Byte ( R[n]+disp);
  assignbit=(0x00000001<<imm)&temp;
  if(assignbit==0) T=1;
  else T=0;
  PC+=4;
}
```

(3) 使用例

```
BLDNOT.B#H'5, @(2, R0)                ;执行前  @(R0+2)=H'20, T=1
                                      ; 执行后  @(R0+2)=H'20, T=0
```

6.3.6 BOR Bit OR 位操作指令

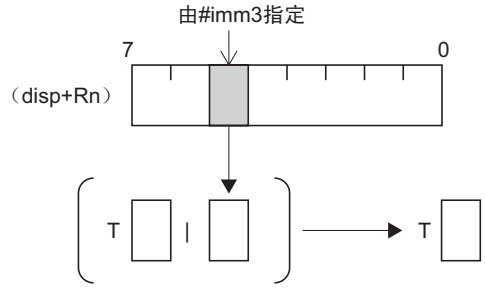
位或 SH-2A/SH2A- FPU （新）

格式	操作概略	操作码	执行状态	T 位
BOR.B #imm3,@(disp12,,Rn)	(<imm>of （disp+Rn）)	T→T	0011nnnn0iii1001 0101dddddddddddd	3	运算结果

(1) 说明

取 (disp+Rn) 所示地址的存储器中指定的 1 位与 T 位的逻辑或，并将其结果保存到 T 位。位号码由 3 位的立即数指定。在本指令下以字节为单位从存储器读取数据。

```
BOR.B #imm3, @(disp12, Rn)
```



(2) 操作内容

```
BORM (long d, long i, long n)  /*BOR.B #imm3, @(disp12, Rn) */
 {
   long disp, imm, temp, assignbit;

     disp = (0x00000FFF & (long)d );
     imm= (0x00000007&(long)i);
     temp= (long) Read_Byte ( R[n]+disp);
     assignbit =(0x00000001<<imm)&temp;
     if((T==0)&&(assignbit==0)) T=0;
     else T=1;

     PC+=4;
}
```

(3) 使用例

```
BOR.B#H'5, @(2, R0)                    ;执行前  @(R0, 2)=H'20, T=0
                                       ;执行后  @(R0, 2)=H'20, T=1
```

6.3.7 BORNOT Bit ORNOT 位操作指令

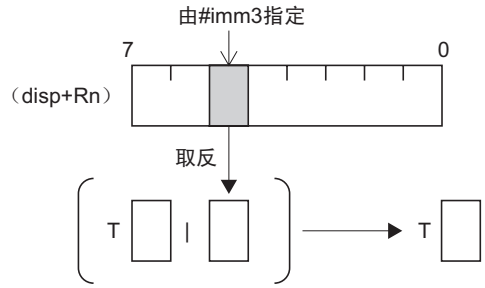
位或非 SH-2A/SH2A-FPU （新）

格式	操作概略	操作码	执行状态	T 位
BORNOT.B #imm3,@(disp12,Rn)	~(<imm>of （disp+Rn）)	T→T	0011nnnn0iii1001 1101dddddddddddd	3	运算结果

(1) 说明

取 (disp+Rn) 所示地址的存储器中指定的 1 位取反后的值与 T 位的逻辑或，并将其结果保存到 T 位。位号码由 3 位的立即数指定。在本指令下以字节为单位从存储器读取数据。

```
BORNOT.B #imm3, @(disp12, Rn)
```



(2) 操作内容

```
BORNOTM (long d, long i, long n)  /*BORNOT.B #imm3, @(disp12, Rn) */
 {
   long disp, imm, temp, assignbit;

     disp = (0x00000FFF & (long)d );
     imm= (0x00000007&(long)i);
     temp= (long) Read_Byte ( R[n]+disp);
     assignbit =(0x00000001<<imm)&temp;
     if((T==1)||(assignbit==0)) T=1;
     else T=0;

     PC+=4;
}
```

(3) 使用例

```
BORNOT.B#H'5, @(2, R0)                ;执行前  @(R0+2)=H'DF, T=0
                                      ;执行后  @(R0+2)=H'DF, T=1
```

6.3.8 BSET Bit SET 位操作指令

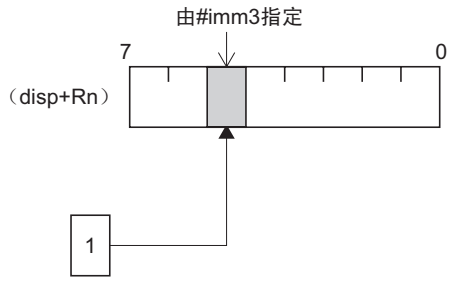
置位 SH-2A/SH2A-FPU （新）

格式	操作概略	操作码	执行状态	T 位
BSET.B #imm3,@(disp12,Rn)	1→(<imm>of （disp+Rn）)	0011nnnn0iii1001 0001dddddddddddd	3	—
BSET #imm3, Rn	1→<imm>of Rn	10000110nnnn1iii	1	—

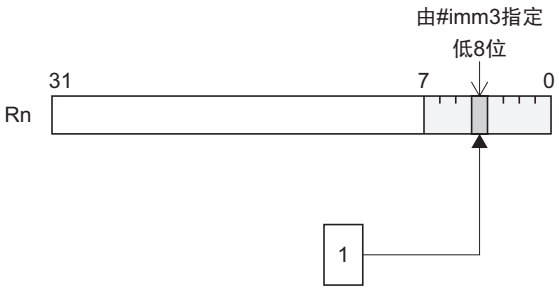
(1) 说明

将 (disp+Rn) 所示地址的存储器或者通用寄存器 Rn 的 LSB 侧 8 位中指定的 1 个位置为 1。位号码由 3 位的立即数指定。在 BSET.B 指令下，以字节为单位从存储器读取数据后，将指定位置 1，并将执行后的数据以字节为单位写入存储器。

```
BSET.B #imm3, @(disp12, Rn)
```




```
BSET #imm3, Rn
```



(2) 操作内容

```
BSETM (long d, long i, long n)  /*BSET.B #imm3, @(disp12, Rn) */
 {
   long disp, imm, temp;

     disp = (0x00000FFF & (long)d );
     imm= (0x00000007&(long)i);
     temp= (long) Read_Byte ( R[n]+disp);
     temp|=(0x00000001<<imm);
     Write_Byte (R[n]+disp, temp);
     PC+=4;
}


BSET (long i, long n)  /*BSET #imm3, Rn */
 {
   long imm, temp;

   imm= (0x00000007 &(long)i);
   R[n]|=(0x00000001<<imm);
  PC+=2;
  }
```

(3) 使用例

```
BSET.B#H'5, @(2, R0)                ;执行前  @(R0+2)=H'00
                                    ;执行后  @(R0+2)=H'20
BSET#H'4, R0                        ;执行前  R0=H'00000000
                                    ;执行后  R0=H'00000010
```

6.3.9 BST Bit STore 位操作指令

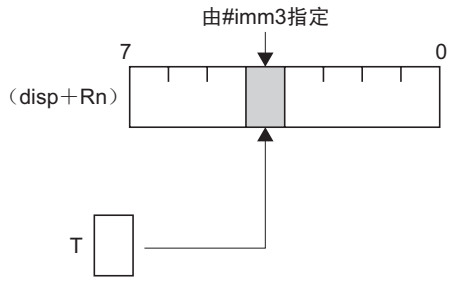
位存储 SH-2A/SH2A-FPU （新）

格式	操作概略	操作码	执行状态	T 位
BST.B #imm3,@(disp12,Rn)	T→(<imm>of （disp+Rn）)	0011nnnn0iii1001 0010dddddddddddd	3	—
BST #imm3, Rn	T→<imm>of Rn	10000111nnnn0iii	1	—

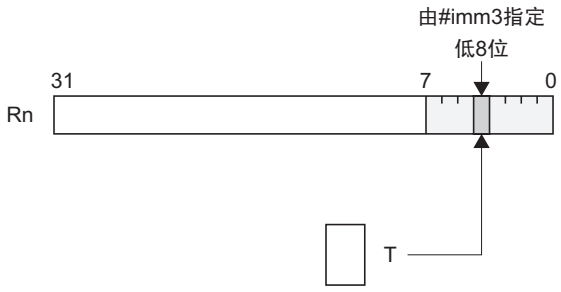
(1) 说明

将 T 位的内容传送到 (disp+Rn) 所示地址的存储器或者通用寄存器 Rn 的 LSB 侧 8 位中指定的 1 位的位置。位号码由 3 位的立即数指定。在 BST.B 命令下，以字节为单位从存储器读取数据后，执行从 T 位到指定位的传送，并将执行后的数据以字节为单位写入存储器。

```
BST.B #imm3, @(disp12, Rn)
```




```
BST #imm3, Rn
```



(2) 操作内容

```
BSTM (long d, long i, long n)  /*BST.B #imm3, @(disp12, Rn) */
 {
   long disp, imm, temp;

  disp = (0x00000FFF & (long)d );
  imm= (0x00000007&(long)i);
  temp = (long) Read_Byte ( R[n]+disp);
if(T==0) temp& = (~(0x00000001<<imm));
else  temp|=(0x00000001<<imm);
  Write_Byte (R[n]+disp, temp);

  PC+=4;

}

BST (long i, long n)  /*BST  #imm3, Rn */
 {
   long disp, imm;

  disp = (0x00000FFF & (long)d );
  imm= (0x00000007&(long)i);
  if (T==0) R[n]& = (~(0x00000001<<imm));
else  R[n]|=(0x00000001<<imm);

PC+=2;
}
```

(3) 使用例

```
BST.B#H'4, @(2, R0)                    ;执行前  @(R0+2)=H'FF, T=0
                                       ;执行后  @(R0+2)=H'EF, T=0
BST#H'4, R0                            ;执行前  R0=H'00000000, T=1
                                       ;执行后  R0=H'00000010, T=1
```

6.3.10 BXOR Bit exclusive OR 位操作指令

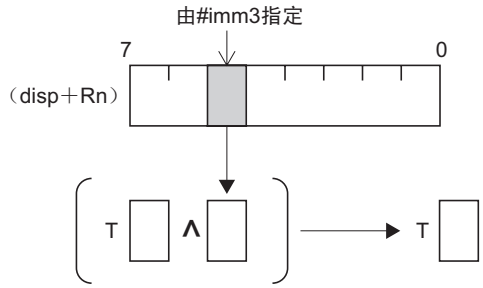
位异或 SH-2A/SH2A-FPU （新）

格式	操作概略	操作码	执行状态	T 位
BXOR.B #imm3,@(disp12,Rn)	(<imm>of （disp+Rn）)^T→T	0011nnnn0iii1001 0110dddddddddddd	3	运算结果

(1) 说明

取 (disp+Rn) 所示地址的存储器指定的 1 位与 T 位的逻辑异或，并将其结果保存到 T 位。位编码由 3 位的立即数指定。在本指令下以字节为单位从存储器读取数据。

```
BXOR.B #imm3, @(disp12, Rn)
```



(2) 操作内容

```
BXORM (long d, long i, long n)  /*BXOR.B #imm3, @(disp12, Rn) */
 {
   long disp, imm, temp, assignbit;

     disp = (0x00000FFF & (long)d );
     imm= (0x00000007&(long)i);
     temp= (long) Read_Byte ( R[n]+disp);
     assignbit =(0x00000001<<imm)&temp;
     if (assignbit==0)
     {
     if(T==0) T=0;
     else  T=1;
     }
      else
      {
      if(T==0) T=1;
     else          T=0;
 }
     PC+=4;
}
```

(3) 使用例

```
BXOR.B#H'5, @(2, R0)                    ;执行前  @(R0+2)=H'FF, T=1
                                        ;执行后  @(R0+2)=H'EF, T=0
```

6.3.11 CLIPS CLIP as Signed 算术运算指令

带符号的饱和值比较指令 SH-2A/SH2A-FPU （新）

No.	格式	操作概略	操作码	执行状态	T 位
1	CLIPS.B Rn	Rn ＞ (饱和上限值) 时，(饱和上限值)→Rn, 1→CS	0100nnnn10010001	1	—
2	CLIPS.W Rn	Rn ＜ (饱和下限值) 时，(饱和下限值)→Rn, 1→CS	0100nnnn10010101	1	—

(1) 说明

判定饱和的指令。在本指令下使用带符号的数据。通用寄存器 Rn 高于饱和上限值或者低于饱和下限值时分别将饱和上限值或者饱和下限值保存到 Rn，并将 CS 位置 1。各指令的饱和上限值、饱和下限值如下表所示。

No.	指令	饱和下限值	饱和上限值
1	CLIPS.B Rn	H'FFFFFF80	H'0000007F
2	CLIPS.W Rn	H'FFFF8000	H'00007FFF

(2) 注意

通用寄存器 Rn 不超过饱和上限值或者不低于饱和下限值时， CS 位的值不变。

(3) 操作内容

```
CLIPSB(long n)  /* CLIPS.B Rn*/
  {
   if ( R[n] > 0x0000007F)
    {
      R[n]=0x0000007F;
      CS=1;
    }
   else if (R[n] < 0xFFFFFF80)
    {
      R[n]=0xFFFFFF80;
      CS=1;
     }
   PC+2;
  }

CLIPSW(long n)  /* CLIPS.W Rn*/
  {
   if ( R[n] > 0x00007FFF)
     {
      R[n]=0x00007FFF;
      CS=1;
      }
   else  if (R[n] < 0xFFFF8000)
      {
        R[n]=0xFFFF8000;
        CS=1;
   PC+2;
  }
```

(4) 使用例

```
CLIPS.B R0                          ;执行前  R0=H'0000000F, CS=0
                                    ;执行后  R0=H'0000000F, CS=0
CLIPS.B R1                          ;执行前  R1=H'00000080, CS=0
                                    ;执行后  R1=H'0000007F, CS=1
CLIPS.W R0                          ;执行前  R0=H'FFFFFFF0, CS=0
                                    ;执行后  R0=H'FFFFFFF0, CS=0
CLIPS.W R1                          ;执行前  R1=H'FFFF7000, CS=0
                                    ;执行后  R1=H'FFFF8000, CS=1
```

6.3.12 CLIPU CLIP as Unsigned 算术运算指令

无符号的饱和值比较指令 SH-2A/SH2A-FPU （新）

No.	格式	操作概略	操作码	执行状态	T 位
1	CLIPU.B Rn	Rn ＞ (饱和值) 时，	0100nnnn10000001	1	—
2	CLIPU.W Rn	(饱和值)→Rn, 1→CS	0100nnnn10000101	1	—

(1) 说明

判定饱和的指令。在本指令下使用无符号的数据。通用寄存器 Rn 超过饱和值时将饱和值保存到 Rn，并将 CS 位置 1。各指令的饱和值如下表所示。

No.	指令	饱和值
1	CLIPU.B Rn	H'000000FF
2	CLIPU.W Rn	H'0000FFFF

(2) 注意

通用寄存器 Rn 不超过饱和上限值时， CS 位的值不变。

(3) 操作内容

```
CLIPUB(long n)  /* CLIPU.B Rn*/
  {
   if ( R[n] > 0x000000FF)
     {
      R[n]=0x000000FF;
      CS=1;
     }
   PC+2;
  }

CLIPUW(long n)  /* CLIPU.W Rn*/
  {
   if ( R[n] > 0x0000FFFF)
     {
      R[n]=0x0000FFFF;
      CS=1;
     }
   PC+2;
  }
```

(4) 使用例

```
CLIPU.B R0                    ;执行前  R0=H'0000000F, CS=0
                              ;执行后  R0=H'0000000F, CS=0
CLIPU.B R1                    ;执行前  R1=H'00000100, CS=0
                              ;执行后  R1=H'000000FF, CS=1
CLIPU.W R0                    ;执行前  R0=H'00000FFF, CS=0
                              ;执行后  R0=H'00000FFF, CS=0
CLIPU.W R1                    ;执行前  R1=H'00010000, CS=0
                              ;执行后  R1=H'0000FFFF, CS=1
```

6.3.13 DIVS DIVide as Signed 算术运算指令

带符号的除法运算 SH-2A/SH2A-FPU （新）

格式	操作概略	操作码	执行状态	T 位
DIVS R0,Rn	带符号 Rn÷R0→Rn	0100nnnn10010100	36	—

(1) 说明

是执行通用寄存器 Rn 的 32 位内容 （被除数）除以 R0 的内容 （除数）的除法运算的指令。本指令执行带符号的除法运算，单独求商。未准备余数的运算。用被除数减去除数和所得商的乘积后求得余数。此时所求得余数的符号与被除数的符号相同。

(2) 注意

负的最大值 （H'80000000）除以 -1 时发生上溢异常。被零除时发生被零除异常。本指令执行过程中，如果产生中断，则中止执行。返回地址成为本指令的起始地址，并再次执行本指令。

(1) 操作内容

```
DIVS  (long n)  /*  DIVS  R0, Rn */
{
  R[n]=R[n] / R[0];
   PC+=2;
 }
```

(2) 使用例

```
 DIVS R0, R1                  ;R1(32 位 )÷R0(32 位 )=R1(32 位 )：带符号
```

6.3.14 DIVU DIVide as Unsigned 算术运算指令

无符号的除法运算 SH-2A/SH2A-FPU （新）

格式	操作概略	操作码	执行状态	T 位
DIVU R0, Rn	无符号 Rn÷R0→Rn	0100nnnn10000100	34	—

(1) 说明

是执行通用寄存器 Rn 的 32 位内容 （被除数）除以 R0 内容 （除数）的除法运算的指令。本指令执行无符号的除法运算，单独求商。未准备余数的运算。用被除数减去除数和所得商的乘积后求得余数。

(2) 注意

被零除时发生被零除异常。

在本指令执行过程中，如果产生中断，则中止执行。返回地址成为本指令的起始地址，并再次执行本指令。

(3) 操作内容

```
DIVU  (long n)  /*  DIVU  R0, Rn */
{
 (unsigned long) R[n]= ( unsigned long )R[n] / (unsigned long )R[0];
 PC+=2;
}
```

(4) 使用例

```
DIVU R0, R1                              ;R1(32 位 )÷R0(32 位 )=R1(32 位 )：无符号
```

6.3.15 FMOV Floating-point MOVe 浮点指令

浮点传送 SH-2A/SH2A-FPU （新）

No.	SZ	格式	操作概略	操作码	执行状态	T 位
1	0	FMOV.S FRm, @(disp12,Rn)	FRm→(disp×4+Rn)	0011nnnnmmmm0001 0011dddddddddddd	1	—
2	1	FMOV.D DRm, @(disp12,Rn)	DRm→(disp×8+Rn)	0011nnnnmmm00001 0011dddddddddddd	2	—
3	0	FMOV.S @(disp12,Rm),FRn	(disp×4+Rm)→FRn	0011nnnnmmmm0001 0111dddddddddddd	1	—
4	1	FMOV.D @(disp12,Rm),DRn	(disp×8+Rm)→DRn	0011nnn0mmmm0001 0111dddddddddddd	2	—

(1) 说明

1. 将FRm的内容传送到(disp+Rn)所示地址的存储器。
2. 将DRm的内容传送到(disp+Rn)所示地址的存储器。
3. 将(disp+Rn)所示地址的存储器内容传送到FRn。
4. 将(disp+Rn)所示地址的存储器内容传送到DRn。

(2) 注意

瑞萨科技的 “SuperH RISC engine 汇编程序”中，必须使用被放大的值 （×4， ×8）表述位移量的值 。

(3) 操作内容

```
void FMOV_INDEX_DISP12_STORE(int  m,n)  /*FMOV.S FRm, @(disp12,Rn) */
{
    long disp;

    disp = (0x00000FFF & (long)d);
     Write_Int ( R[n]+(disp<<2), FR[m]);
     PC +=4;
}

void FMOV_INDEX_DISP12_STORE_DR(int  m,n)
/*FMOV.D DRm, @(disp12,Rn) */
{
    long disp;

    disp = (0x00000FFF & (long)d);
     Write_Quad ( R[n]+(disp<<3), DR[m>>1]);
     PC +=4;
}
```

```
void FMOV_INDEX_DISP12_LOAD(int  m,n)  /*FMOV.S @(disp12,Rm), FRn */
{
    long disp;

    disp = (0x00000FFF & (long)d);
     FR[n] = Read_Int ( R[m]+(disp<<2));
     PC +=4;
}

void FMOV_INDEX_DISP12_LOAD_DR(int m,n)
                                                /*FMOV.D @(disp12,Rm), DRn */
{
    long disp;

    disp = (0x00000FFF & (long)d);
     DR[n>>1] = Read_Quad ( R[m]+(disp<<3));
     PC +=4;
}
```

(4) 使用例

```
FMOV.S FR0, @(2, R2)                ;执行前  FR0=H'12345670
                                    ;执行后  @(R2+8)=H'12345670
FMOV.D DR0, @(2, R2)                ;执行前  FR0=H'01234567
                                    ;        FR1=H'89ABCDEF
                                    ;执行后  @(R2+16)=H'01234567
                                    ;        @(R2+20)=H'89ABCDEF
FMOV.S @(2, R2), FR0                ;执行前  @(R2+8)=H'12345670
                                    ;执行后  FR0=H'12345670
FMOV.D @(2, R2), DR0                ;执行前  @(R2+16)=H'01234567
                                    ;        @(R2+20)=H'89ABCDEF
                                    ;执行后  FR0=H'01234567
                                    ;        FR1=H'89ABCDEF
```

6.3.16 JSR/N Jump to SubRoutine with No delay slot 转移指令

向无延迟槽子程序过程的转移 SH-2A/SH2A-FPU （新）

格式	操作概略	操作码	执行状态	T 位
JSR/N @Rm	PC-2→PR,Rm→PC	0100mmmm01001011	3	—
JSR/N @@(disp8, TBR)	PC-2→PR, (disp×4+TBR)→PC	10000011dddddddd	5	—

(1) 说明

转移到指定地址的子程序过程。在将 PC 的内容保存到 PR 后，转移到通用寄存器 Rm 的 32 位内容所指示的地址或者是从 (disp×4+TBR) 地址的存储器读取的地址。被保存的 PC 为本指令的 2 个地址后的起始地址。与 RTS 的组合用于子程序过程的调用。

(2) 注意

本指令不是延迟转移。

瑞萨科技的 “SuperH RISC engine 汇编程序”中，必须使用被放大的值 （×4）表述位移量的值。

(3) 操作内容

```
JSRN (long m)   /* JSR/N  @Rm, */
{
   unsigned  long  temp;

   temp=PC;
   PR=PC-2;
   PC=R[m]+4;
}

JSRNM (long d )   /* JSR/N  @@(disp8, TBR) */
{
   unsigned  long  temp;
   long disp;

   temp=PC;
   PR=PC-2;
   disp=(0x000000FF & d);
   PC=Read_Long(TBR+(disp<<2))+4;
}
```

(4) 使用例

```
                MOV.L JSRN_TABLE, R0;R0=TRGET 的地址
                JSR/N @R0             ; 转移到 TRGET。
                ADD    R0, R1         ;← 从过程返回的地址 （PR 的内容）。
                . . . . . . . . .
                .align 4
JSRN_TABLE:     .data.1 TRGET         ; 跳转表
TRGET:          NOP                   ;← 过程的入口
                MOV   R2, R3          ;
                RTS/N                 ; 返回到上述 ADD 指令。


TBR+H'08        .data.1  FFFF7F80     ;
                . . . . . . . . . .
                JSR/N  @@(2, TBR)     ; 转移到地址 TBR+H'08 内容的地址。
                ADD    R0, R1         ; ← 从过程返回的地址 （PR 的内容）。
                . . . . . . . . . .
FFFF7F80        NOP                   ;← 过程的入口
FFFF7F82        MOV   R2, R3          ;
FFFF7F84        RTS/N                 ; 返回到 ADD 指令。
```

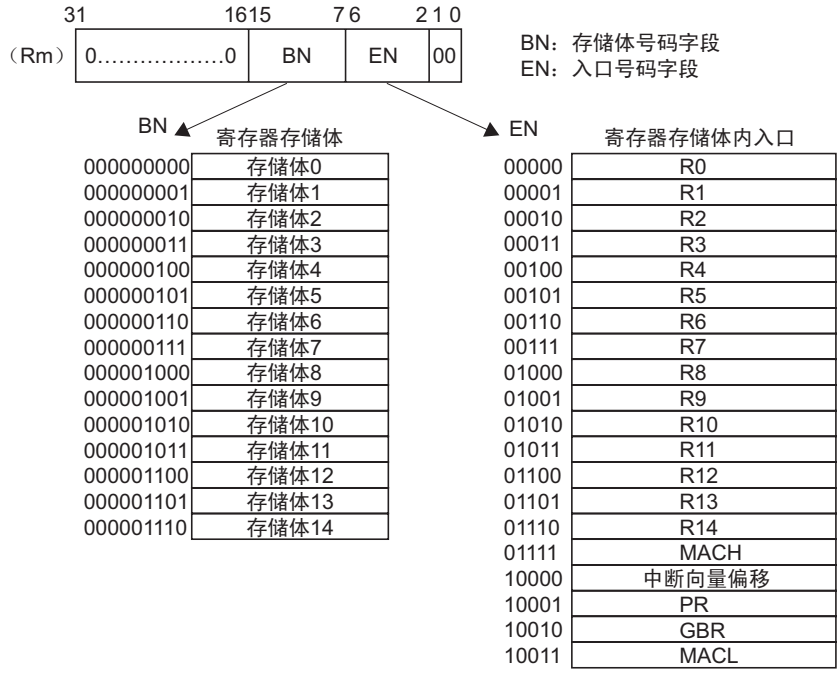
6.3.17 LDBANK LoaD register BANK 系统控制指令

传送至指定寄存器存储体的入口 SH-2A/SH2A-FPU （新）

格式	操作概略	操作码	执行状态	T 位
LDBANK @Rm, R0	（指定寄存器存储体入口） → R0	0100mmmm11100101	6	—

(1) 说明

将通用寄存器 Rm 内容所指定的寄存器存储体中的 1 个入口传送到通用寄存器 R0。通用寄存器 Rm 指定寄存器存储体的号码及存储体内保存的寄存器。



(2) 注意

在体系结构上，最多可持有 512 个存储体。
但是，存储体的数量因产品而异。

(3) 操作内容

```
LDBANK ( long m)  /*LDBANK  @Rm, R0 */
{
R[0]=Read_Bank_Long( R[m] );
PC+=2;
}
```

(4) 使用例

```
LDBANK@R1, R0                    ;执行前  R1=H'00000108
                                 ;执行后  R0= 保存到存储体 2 的 R2 的内容
```

6.3.18 LDC LoaD to Control register 系统控制指令

加载到控制寄存器 SH-2A/SH2A-FPU （新）

格式	操作概略	操作码	执行状态	T 位
LDC Rm, TBR	Rm→TBR	0100mmmm01001010	1	—

(1) 说明

将源操作数保存到控制寄存器 TBR。

(2) 操作内容

```
LDCTBR (long m)  /* LDC Rm, TBR*/
 {
  TBR=R[m];
  PC+=2;
}
```

(3) 使用例

```
LDC R0, TBR                      ;执行前  R0=H'12345678, TBR=H'00000000
                                 ;执行后  TBR=H'12345678
```

6.3.19 MOV MOVe structure data 数据传送指令

结构体数据的传送 SH-2A/SH2A-FPU （新）

格式	操作概略	操作码	执行状态	T 位
MOV.B Rm, @(disp12,Rn)	Rm→(disp+Rn)	0011nnnnmmmm0001 0000dddddddddddd	1	—
MOV.W Rm, @(disp12,Rn)	Rm→(disp×2+Rn)	0011nnnnmmmm0001 0001dddddddddddd	1	—
MOV.L Rm, @(disp12,Rn)	Rm→(disp×4+Rn)	0011nnnnmmmm0001 0010dddddddddddd	1	—
MOV.B @(disp12,Rm), Rn	(disp+Rm)→ 符号扩展 →Rn	0011nnnnmmmm0001 0100dddddddddddd	1	—
MOV.W @(disp12,Rm), Rn	(disp×2+Rm)→ 符号扩展 →Rn	0011nnnnmmmm0001 0101dddddddddddd	1	—
MOV.L @(disp12,Rm), Rn	(disp×4+Rm)→Rn	0011nnnnmmmm0001 0110dddddddddddd	1	—

(1) 说明

将源操作数传送到目的地。最适合结构体，堆栈内的数据存取。

(2) 注意

瑞萨科技的 “SuperH RISC engine 汇编程序”中，必须使用被放大的值 （×1、 ×2、 ×4）表述位移量的值。

(3) 操作内容

```
MOVBS12 (long d, long m, long n)   /* MOV.B Rm, @(disp12,Rn) */
{
   long disp;

   disp = (0x00000FFF & (long)d);
   Write_Byte(R[n]+disp,R[m]);
   PC+=4;
}

MOVWS12 (long d, long m, long n)   /* MOV.W Rm, @(disp12,Rn) */
{
   long disp;

   disp = (0x00000FFF & (long)d);
   Write_Word(R[n]+(disp<<1),R[m]);
   PC+=4;
}
```

```
MOVLS12 (long d, long m, long n)   /* MOV.L Rm, @(disp12,Rn) */
{
   long disp;

   disp = (0x00000FFF & (long)d);
   Write_Long(R[n]+(disp<<2), R[m]);
   PC+=4;
}

MOVBL12 (long d, long m, long n)   /* MOV.B @(disp12,Rm), Rn */
{
   long disp;

   disp = (0x00000FFF & (long)d);
   R[n]=Read_Byte(R[m]+disp);
   if ( ( R[n]&0x80 ) ==0) R[n] &=0x000000FF;
   else R[0] |=0xFFFFFF00;
   PC+=4;
}

MOVWL12 (long d, long m, long n)   /* MOV.W @(disp12,Rm), Rn */
{
   long disp;

   disp = (0x00000FFF & (long)d);
   R[n]=Read_Word(R[m]+(disp<<1));
   if ( ( R[n]&0x8000 ) ==0) R[n] &=0x0000FFFF;
   else R[n] |=0xFFFF0000;
    PC+=4;
}

MOVLL12 (long d, long m, long n)   /* MOV.L @(disp12,Rm), Rn */
{
   long disp;

   disp = (0x00000FFF & (long)d);
   R[n]=Read_Long(R[m]+(disp<<2));
   PC+=4;
}
```

(4) 使用例

```
MOV.B R0, @(1, R1)          ;执行前  R0=H'FFFF7F80
                            ;执行后  @(R1+1)=H'80
MOV.L @(2, R0), R1          ;执行前  @(R0+8)=H'12345670
                            ;执行后  R1=H'12345670
```

6.3.20 MOV MOVe reverse stack 数据传送指令

反向堆栈传送 SH-2A/SH2A-FPU （新）

格式	操作概略	操作码	执行状态	T 位
MOV.B R0, @Rn+	R0→(Rn), Rn+1→Rn	0100nnnn10001011	1	—
MOV.W R0, @Rn+	R0→(Rn), Rn+2→Rn	0100nnnn10011011	1	—
MOV.L R0, @Rn+	R0→(Rn), Rn+4→Rn	0100nnnn10101011	1	—
MOV.B @-Rm, R0	Rm-1→Rm (Rm)→ 符号扩展 →R0	0100mmmm11001011	1	—
MOV.W @-Rm, R0	Rm-2→Rm (Rm)→ 符号扩展 →R0	0100mmmm11011011	1	—
MOV.L @-Rm, R0	Rm-4→Rm (Rm)→R0	0100mmmm11101011	1	—

(1) 说明

将源操作数传送到目的地。

(2) 操作内容

```
MOVRSBP (long n)   /* MOV.B R0, @Rn+*/
{
   Write_Byte(R[n], R[0]);
   R[n]+=1;
   PC+=2;
}

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

MOVRSLP (long n)   /* MOV.L R0, @Rn+*/
{
   Write_Long(R[n], R[0]);
   R[n]+=4;
   PC+=2;
}
MOVRSBM (long m)   /* MOV.B @-Rm, R0*/
{
R[m] － =1;
   R[0]=(long) Read_Word (R[m]);
   if ( (R[0]&0x80)==0)  R[0]&=0x000000FF;
```

```
   else R[0] |=0xFFFFFF00;

   PC+=2;
}

MOVRSWM (long m)   /* MOV.W @-Rm, R0*/
{
R[m] ― =2;
   R[0]=(long) Read_Word (R[m]);
   if ( (R[0]&0x8000)==0)  R[0]&=0x0000FFFF;
   else R[0] |=0xFFFF0000;

   PC+=2;
}

MOVRSLM(long m)   /* MOV.L @-Rm, R0*/
{
R[m] ― =4;
R[0]=Read_Long (R[m]);

    PC+=2;
}
```

(3) 使用例

```
MOV.B R0, @R1+              ; 执行前  R0=H'AAAAAAAA, R1=H'FFFF7F80
                            ; 执行后  R1=H'FFFF7F81, @(H'FFFF7F80)=H'AA
MOV.L @-R1, R0              ; 执行前  R1=H'12345678
                            ; 执行后  R1=H'12345674, R0=@(H'12345674)
```

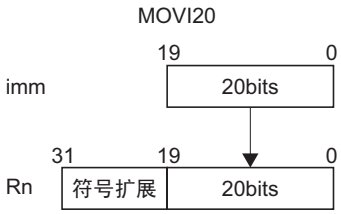
6.3.21 MOVI20 MOVe Immediate 20bits data 数据传送指令

20 位立即数的传送 SH-2A/SH2A-FPU （新）

格式	操作概略	操作码	执行状态	T 位
MOVI20 #imm20, Rn	imm→ 符号扩展 →Rn	0000nnnniiii0000 iiiiiiiiiiiiiiii	1	—

(1) 说明

将符号扩展为长字的立即数保存到通用寄存器 Rn。



(2) 操作内容

```
MOVI20 ( long i, long n)    /* MOVI20 #imm, Rn */
 {
        if ( i&0x00080000 ) ==0) R[n]= (0x000FFFFF & (long) i );
        else R[n]=(0xFFF00000 | (long) i );

    PC+=4;
   }
```

(3) 使用例

```
MOVI20 H'7FFFF, R0          ;执行前  R0=H'00000000
                            ;执行后  R0=H'0007FFFF
```

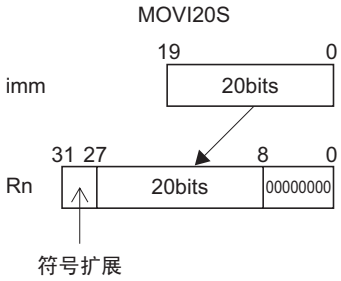
6.3.22 MOVI20S MOVe Immediate 20bits data and 8bits Shift left 数据传送指令

20 位立即数的传送 / 左移 8 位 SH-2A/SH2A-FPU （新）

格式	操作概略	操作码	执行状态	T 位
MOVI20S #imm20, Rn	imm<<8 → 符号扩展 →Rn	0000nnnniiii0001 iiiiiiiiiiiiiiii	1	—

(1) 说明

将立即数左移 8 位，符号扩展为长字后，保存到通用寄存器 Rn。通过使用作为后续指令的 ADD 指令或者 OR 指令，可以生成 28 位的绝对地址。详细内容请参照 “附录 B. 编程的指针 (关于 MOVI20、 MOVI20S 的使用方法)”。



(2) 注意

瑞萨科技的 “SuperH RISC engine 汇编程序”中，必须将立即数左移 8 位。

(3) 操作内容

```
MOVI20S (long i, long n)    /* MOVI20S #imm, Rn */
 {
    if ( i&0x00080000 ) ==0) R[n]= (0x000FFFFF & (long) i );
    else R[n]=(0xFFF00000 | (long) i );
    R[n]<<=8;
    PC+=4;
 }
```

(4) 使用例

```
MOVI20S H'7FFFF, R0          ; 执行前  R0=H'00000000
                             ; 执行后  R0=H'07FFFF00
```

6.3.23 MOVML.L MOVe Multi-register Lower part 数据传送指令

R0 ～ Rn 寄存器的保存 / 恢复指令 SH-2A/SH2A-FPU （新）

格式	操作概略	操作码	执行状态	T 位
MOVML.L Rm, @-R15	R15-4→R15, Rm→(R15) R15-4→R15, Rm-1→(R15) ・ ・ R15-4→R15, R0→(R15) ※ Rm=R15 时， Rm 读作 PR	0100mmmm11110001	1 ～ 16	—
MOVML.L @R15+, Rn	(R15)→R0, R15+4→R15 (R15)→R1, R15+4→R15 ・ ・ (R15)→Rn, R15+4→R15 ※ Rn=R15 时， Rn 读作 PR	0100nnnn11110101	1 ～ 16	—

(1) 说明

将源操作数传送到目的地。该指令传送小于等于指定寄存器编码的多个通用寄存器 (R0 ～ Rn/Rm) 与将 R15 的内容作为地址的存储器。

指定 R15 时，代替 R15 传送 PR。即指定 nnnn(mmmm)=1111 时， R0 ～ R14、 PR 成为传送对象的通用寄存器。

(2) 操作内容

```
MOVLMML (long m) /*MOVML.L Rm, @-R15*/
{
   long i;

  for (i=m; i ≥ 0; i －－ )
   {
     if (i==15)
      {
        Write_Long (R[15] － 4, PR);
        R[15] －＝ 4;
      }
    else
     {
        Write_Long (R[15] － 4, R[i]);
        R[15] －＝ 4;
      }
   }

 PC+=2;
 }

MOVLPML (long n )  /*MOVML.L @R15+, Rn */
{
  int  i;

  for ( i=0; i ≦ n; i++ )
   {
     if (i==15)
        {
          PR=Read_Long (R[15]);
         }
     else
        {
        R[i] = Read_Long (R[15]);
         }
      R[15]+=4;
   }
  PC+=2;
}
```

(3) 使用例

```
MOVML.L R7, @-R15          ;执行前  R15=H'FFFF7F80
                                    R0=H'00000000, R1=H'11111111
                                    R2=H'22222222, R3=H'33333333
                                    R4=H'44444444, R5=H'55555555
                                    R6=H'66666666, R7=H'77777777
                           ;执行后  R15=H'FFFF7F60
                                    @(H'FFFF7F7C)=H'77777777
                                    @(H'FFFF7F78)=H'66666666
                                    @(H'FFFF7F74)=H'55555555
                                    @(H'FFFF7F70)=H'44444444
                                    @(H'FFFF7F6C)=H'33333333
                                    @(H'FFFF7F68)=H'22222222
                                    @(H'FFFF7F64)=H'11111111
                                    @(H'FFFF7F60)=H'00000000
MOVML.L @R15, R7           ;执行前  R15=H'FFFF7F60
                                    @(H'FFFF7F60)=H'00000000
                                    @(H'FFFF7F64)=H'11111111
                                    @(H'FFFF7F68)=H'22222222
                                    @(H'FFFF7F6C)=H'33333333
                                    @(H'FFFF7F70)=H'44444444
                                    @(H'FFFF7F74)=H'55555555
                                    @(H'FFFF7F78)=H'66666666
                                    @(H'FFFF7F7C)=H'77777777
                           ;执行后  R15=H'FFFF7F80
                                    R0=H'00000000, R1=H'11111111
                                    R2=H'22222222, R3=H'33333333
                                    R4=H'44444444, R5=H'55555555
                                    R6=H'66666666, R7=H'77777777
```

6.3.24 MOVMU.L MOVe Multi-register Upper part 数据传送指令

Rn ～ R14， PR 寄存器的保存 / 恢复指令 SH-2A/SH2A-FPU （新）

格式	操作概略	操作码	执行状态	T 位
MOVMU.L Rm, @-R15	R15-4→R15, PR→(R15) R15-4→R15, R14→(R15) ・ ・ R15-4→R15, Rm→(R15) ※ Rm=R15 时， Rm 读作 PR	0100mmmm11110000	1 ～ 16	—
MOVMU.L @R15+, Rn	(R15)→Rn,R15+4→R15 (R15)→Rn+1,R15+4→R15 ・ (R15)→R14, R15+4→R15 (R15)→PR, R15+4→R15 ※ Rn=R15 时， Rn 读作 PR	0100nnnn11110100	1 ～ 16	—

(1) 说明

将源操作数传送到目的地。该指令传送大于等于指定寄存器编码的多个通用寄存器 （Rn/Rm ～ R14，PR）与将 R15 的内容作为地址的存储器。

指定 R15 时，代替 R15 传送 PR。

(2) 操作内容

```
MOVLMMU (long m) /*MOVMU.L Rm, @－R15 */
{
   int i;

   Write_Long (R[15]－4, PR);
   R[15]－＝4;

   for ( i = 14; i≧m; i－－ )
    {
    Write_Long (R[15]－4, R[i]);
    R[15]－＝4;
    }
 PC+=2;
 }


MOVLPMU (long n)  /*MOVMU.L @R15+, Rn*/
    {
      int i;
```

```
    for ( i=n; i ≦ 14; i++ )
      {
       R[i] = Read_Long (R[15]);
       R[15]+=4;
      }
    PR=Read_Long (R[15]);
    R[15]+=4;
    PC+=2;
}
```

(3) 使用例

```
MOVMU.L R8, @-R15      ; 执行前   R15=H'FFFF7F80
                                  R8=H'88888888, R9=H'999999999
                                  R10=H'AAAAAAAA, R11=H'BBBBBBBB
                                  R12=H'CCCCCCCC, R13=H'DDDDDDDD
                                  R14=H'EEEEEEEE, PR=H'FFFFFFF0
                       ; 执行后   R15=H'FFFF7F60
                                  @(H'FFFF7F7C)=H'FFFFFFF0
                                  @(H'FFFF7F78)=H'EEEEEEEE
                                  @(H'FFFF7F74)=H'DDDDDDDD
                                  @(H'FFFF7F70)=H'CCCCCCCC
                                  @(H'FFFF7F6C)=H'BBBBBBBB
                                  @(H'FFFF7F68)=H'AAAAAAAA
                                  @(H'FFFF7F64)=H'99999999
                                  @(H'FFFF7F60)=H'88888888

MOVMU.L @R15, R8       ; 执行前   R15=H'FFFF7F60
                                  @(H'FFFF7F60)=H'88888888
                                  @(H'FFFF7F64)=H'99999999
                                  @(H'FFFF7F68)=H'AAAAAAAA
                                  @(H'FFFF7F6C)=H'BBBBBBBB
                                  @(H'FFFF7F70)=H'CCCCCCCC
                                  @(H'FFFF7F74)=H'DDDDDDDD
                                  @(H'FFFF7F78)=H'EEEEEEEE
                                  @(H'FFFF7F7C)=H'FFFFFFF0
                       ; 执行后   R15=H'FFFF7F80
                                  R8=H'88888888, R9=H'99999999
                                  R10=H'AAAAAAAA, R11=H'BBBBBBBB
                                  R12=H'CCCCCCCC, R13=H'DDDDDDDD
                                  R14=H'EEEEEEEE, PR=H'FFFFFFF0
```

6.3.25 MOVRT MOVe Reverse Tbit 数据传送指令

T 位取反传送 Rn SH-2A/SH2A-FPU （新）

格式	操作概略	操作码	执行状态	T 位
MOVRT Rn	~T→Rn	0000nnnn00111001	1	—

(1) 说明

T 位取反后，将结果保存到通用寄存器 Rn。 T=1 时 Rn=0， T=0 时 Rn=1。

(2) 操作内容

```
MOVRT ( long n )  /*MOVRT Rn */
    {
     if (T ==1) R[n]=0x00000000;
     else R[n] = 0x00000001;
    PC+=2;
    }
```

(3) 使用例

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

6.3.26 MOVU MOVe structure data as Unsigned 数据传送指令

结构体数据的无符号传送 SH-2A/SH2A-FPU （新）

格式	操作概略	操作码	执行状态	T 位
MOVU.B @(disp12,Rm), Rn	(disp+Rm)→ 零扩展 →Rn	0011nnnnmmmm00011 000ddddddddddd	1	—
MOVU.W @(disp12,Rm), Rn	(disp×2+Rm)→ 零扩展 →Rn	0011nnnnmmmm00011 001ddddddddddd	1	—

(1) 说明

将源操作数传送到目的地。本指令进行无符号数据的传送。最适合结构体、堆栈内的数据存取。

(2) 注意

瑞萨科技的 “SuperH RISC engine 汇编程序”中，必须使用被放大的值 （×1， ×2）表述位移量的值。

(3) 操作内容

```
MOVBUL12 (long d, long m, long n)   /* MOVU.B @(disp12,Rm), Rn */
{
   long disp;

   disp = (0x00000FFF & (long)d);
   R[n]=Read_Byte(R[m]+disp);
   R[n] &=0x000000FF;
   PC+=4;
}

MOVWUL12 (long d, long m, long n)   /* MOVU.W @(disp12,Rm), Rn */
{
   long disp;

   disp = (0x00000FFF & (long)d);
   R[n]=Read_Word(R[m]+(disp<<1));
   R[n] &=0x0000FFFF;
   PC+=4;
}
```

(4) 使用例

```
MOVU.B@(2, R0), R1          ;执行前  @(R0+2)=H'FF
                            ;执行后  R1=H'000000FF
MOVU.W@(2, R0), R1          ;执行前  @(R0+4)=H'FFFF
                            ;执行后  R1=H'0000FFFF
```

6.3.27 MULR MULtiply to Register 算术运算指令

带 Rn 结果保存符号的乘法运算 SH-2A/SH2A-FPU （新）

格式	操作概略	操作码	执行状态	T 位
MULR R0,Rn	R0×Rn→Rn	0100nnnn10000000	2	—

(1) 说明

通用寄存器 R0 的内容与 Rn 进行 3 2 位的乘法运算，将结果的低 32 位保存到通用寄存器 Rn。

(2) 操作内容

```
MULR ( long n)   /* MULR  R0, Rn */
 {
     R[n] = R[0]*R[n];
     PC+=2;
 }
```

(3) 使用例

```
MULR R0, R1                          ;执行前  R0=H'FFFFFFFE, R1=H'00005555
                                     ;执行后  R1=H'FFFF5556
```

6.3.28 NOTT NOT Tbit 数据传送指令

T 位取反传送 SH-2A/SH2A-FPU （新）

格式	操作概略	操作码	执行状态	T 位
NOTT	~T→T	0000000001101000	1	运算结果

(1) 说明

取反 T 位后，将结果保存到 T 位。

(2) 操作内容

```
NOTT ( long n )  /*NOTT Rn */
 {
  if (T ==1) T=0;
  else T=1;
 PC+=2;
 }
```

(3) 使用例

```
SETT                    ;T=1
NOTT                    ;T=0
NOTT                    ;T=1
```

6.3.29 PREF PREFetch data to cache 数据传送指令

向数据高速缓存的预取 SH4

格式	操作概略	操作码	执行状态	T 位
PREF @Rn	prefetch cache block	0000nnnn10000011	1	—

(1) 说明

将在 16 字节边界开始的 16 字节数据块读入操作数高速缓存。
该指令不发生有关地址的错误。产生错误时，该指令作为 NOP(无操作) 指令处理。

(2) 注意

不具有高速缓存的产品，作为 NOP 指令处理。

(3) 操作内容

```
PREF ( long n )  /* PREF @Rn */
 {
 PC+=2;
 }
```

(4) 使用例

```
                MOV.L SOFT_PF, R1   ;R1 的地址为 SOFT_PF
                PREF @R1            ; 将 SOFT_PF 的数据加载到内部数据高速缓存

                .align   16
SOFT_PF:        .data.w H'1234
                .data.w H'5678
                .data.w H'9ABC
                .data.w H'DEF0
```

6.3.30 RESBANK REStore from registerBANK 系统控制指令

从寄存器存储体的寄存器返回 SH-2A/SH2A-FPU （新）

格式	操作概略	操作码	执行状态	T 位
RESBANK	从寄存器存储体的返回	0000000001011011	9*	—

【注】 * 发生存储体上溢，且寄存器从堆栈返回时为 19。

(1) 说明

从最后保存的寄存器存储体返回寄存器。

(2) 操作内容

```
RESBANK( )   /*RESBANK */
                /*m=( 最后进行保存的寄存器存储体的号码 )*/
{
 int  m;

if(BO==0)
{
     PR = Register_Bank[m].PR_BANK;
     GBR = Register_Bank[m].GBR_BANK;
     MACL = Register_Bank[m].MACL_BANK;
     MACH = Register_Bank[m].MACH_BANK;
     for (i=14; i ≥ 0; i--)
     {
      R[i] = Register_Bank[m].R_BANK[i];
     }
  }
else
  {
     for (i=0; i ≦ 14; i++)
     {
      R[i] = Read_Long(R[15]);
      R[15]+=4;
     }
     PR=Read_Long(R[15]);
     R[15]+=4;
     GBR=Read_Long(R[15]);
     R[15]+=4;
     MACH=Read_Long(R[15]);
     R[15]+=4;
     MACL =Read_Long(R[15]);
     R[15]+=4;
}
```

```
PC+=2;

}
```

(3) 使用例

```
RESBANK                              ; 从寄存器存储体返回寄存器。
RTE                                  ; 返回到源程序。
ADD                    #8, R14       ; 在转移之前执行。
```

6.3.31 RTS/N ReTurn from Subroutine with No delay slot 转移指令

从无延迟槽子程序过程返回 SH-2A/SH2A-FPU （新）

格式	操作概略	操作码	执行状态	T 位
RTS/N	PR→PC	0000000001101011	3	—

(1) 说明

从子程序过程返回。即：在从 PR 返回 PC 后，从返回的 PC 所指示的地址开始继续进行处理。可通过本指令从 BSR 和 JSR 指令调用的子程序过程返回到调用源。

(2) 注意

本指令并非延迟转移指令。

(3) 操作内容

```
RTSN (  )   /* RTS/N  */
{
   PC=PR+4;
}
```

(4) 使用例

```
                    MOV.L TABLE, R3    ;R0=TRGET 的地址
                    JSR/N @R3          ; 转移到 TRGET。
                    ADD    R0, R1      ;← 从过程返回的地址 （PR 的内容）。
                    . . . . . . . .
TABLE:              .data.1 TRGET      ; 跳转表
                    . . . . . . . .
TRGET:              NOP                ;← 过程的入口
                    MOV   R2, R3       ;
                    RTS/N              ; 返回到上述 ADD 指令。
```

6.3.32 RTV/N ReTurn to Value and from subroutine with No delay slot 转移指令

从带寄存器值传送的无延迟槽子程序过程返回 SH-2A/SH2A-FPU（新）

格式	操作概略	操作码	执行状态	T 位
RTV/N Rm	Rm→R0, PR→PC	0000mmmm01111011	3	—

(1) 说明

从指定通用寄存器 Rm 传送到 R0 后，从子程序过程返回。即：在将 Rm 的值保存到 R0 后， PC 从 PR 返回，并从返回的 PC 所指示的地址开始继续进行处理。可通过本指令从 BSR 和 JSR 指令调用的子程序过程返回到调用源。

(2) 注意

本指令并非延迟转移指令。

(3) 操作内容

```
RTVN ( int m)   /* RTV/N  Rm */
{
   R[0]=R[m];
   PC=PR+4;
 }
```

(4) 使用例

```
             MOV.L TABLE, R3      ;R0=TRGET 的地址
             JSR/N @R3            ; 转移到 TRGET。
             ADD    R0, R1        ;← 从过程返回的地址 （PR 的内容）。
             . . . . . .
TABLE:       .data.1 TRGET        ; 跳转表
             . . . . . .
TRGET:       NOP                  ;← 过程的入口
             MOV   #12, R3        ;R3=H'00000012
             RTV/N R3             ; 返回到上述 ADD 指令。
                                  ;R0=H'00000012
```

6.3.33 SHAD SHift Arithemetic Dynamically 移位指令

动态算术移位

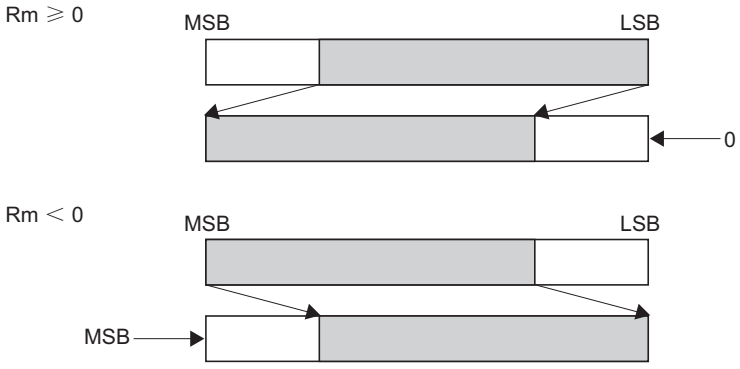
格式	操作概略	操作码	执行状态	T 位
SHAD Rm, Rn	Rm ≥ 0 时， Rn<<Rm→Rn Rm<0 时， Rn>>|Rm|→[MSB→Rn]	0100nnnnmmmm1100	1	—

(1) 说明

将通用寄存器 Rn 的内容算术移位。通用寄存器 Rm 指定移位的方向及移动的位数。

Rm 寄存器的值为正时左移，为负时右移。右移时高位追加 MSB。

移动的位数由 Rm 寄存器的低 5 位 (bit4 ～ 0) 指定。值为负 (MSB=1) 时， Rm 寄存器用 2 的补码表示。因此，右移的移位量为 Rm 寄存器低 5 位的取反加 1 得到的数。左移的移位量为 0 ～ 31，右移的移位量为 1 ～ 32。



(2) 操作内容

```
SHAD (int m,n)  /* SHAD Rm,Rn */
  {
     int  sgn = R[m] & 0x80000000;
     if  (sgn == 0)
              R[n] <<= (R[m] & 0x0000001F);
     else  if  ((R[m] & 0x0000001F) == 0)
         {
          if (( R[n] & 0x80000000) == 0)
                R[n] = 0;
          else
              R[n]=0xFFFFFFFF;
         }
     else
              R[n]=(long)R[n] >> ((~R[m] & 0x0000001F)+1);
     PC+=2;
  }
```

(3) 使用例

```
SHAD R1, R2              ;执行前  R1=H'FFFFFFEC, R2=H'80180000
                         ;执行后  R1=H'FFFFFFEC, R2=H'FFFFF801
SHAD R3, R4              ;执行前  R3=H'00000014, R4=H'FFFFF801
                         ;执行后  R3=H'00000014, R4=H'80100000
```

6.3.34 SHLD SHift Logical Dynamically 移位指令

动态逻辑移位

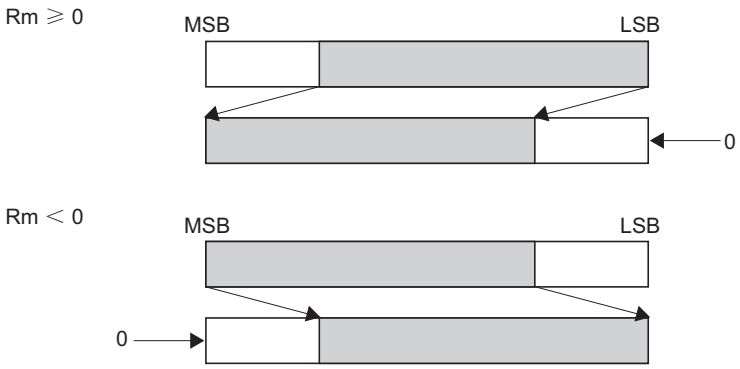
格式	操作概略	操作码	执行状态	T 位
SHLD Rm, Rn	Rm ≥ 0 时， Rn<<Rm→Rn Rm<0 时， Rn>>|Rm|→[0→Rn]	0100nnnnmmmm1101	1	—

(1) 说明

将通用寄存器 Rn 的内容逻辑移位。通用寄存器 Rm 指定移位的方向及移动的位数。

Rm 寄存器的值为正时左移，为负时右移。右移时高位追加 0。

移动的位数由 Rm 寄存器的低 5 位 (bit4 ～ 0) 指定。值为负 (MSB=1) 时， Rm 寄存器用 2 的补码表示。因此，右移的移位量为 Rm 寄存器低 5 位的取反加 1 得到的数。左移的移位量为 0 ～ 31，右移的移位量为 1 ～ 32。



(2) 操作内容

```
SHLD (int m,n)  /* SHLD Rm,Rn */
 {
    int  sgn = R[m] & 0x80000000;
    if  (sgn == 0)
             R[n] <<= (R[m] & 0x0000001F);
    else  if  ((R[m] & 0x0000001F) == 0)
             R[n] = 0;
    else
             R[n]=(unsigned)R[n] >> ((~R[m] & 0x0000001F)+1);
    PC+=2;
 }
```

(3) 使用例

```
SHLD R1, R2                ;执行前  R1=H'FFFFFFEC, R2=H'80180000
                           ;执行后  R1=H'FFFFFFEC, R2=H'00000801
SHLD R3, R4                ;执行前  R3=H'00000014, R4=H'FFFFF801
                           ;执行后  R3=H'00000014, R4=H'80100000
```

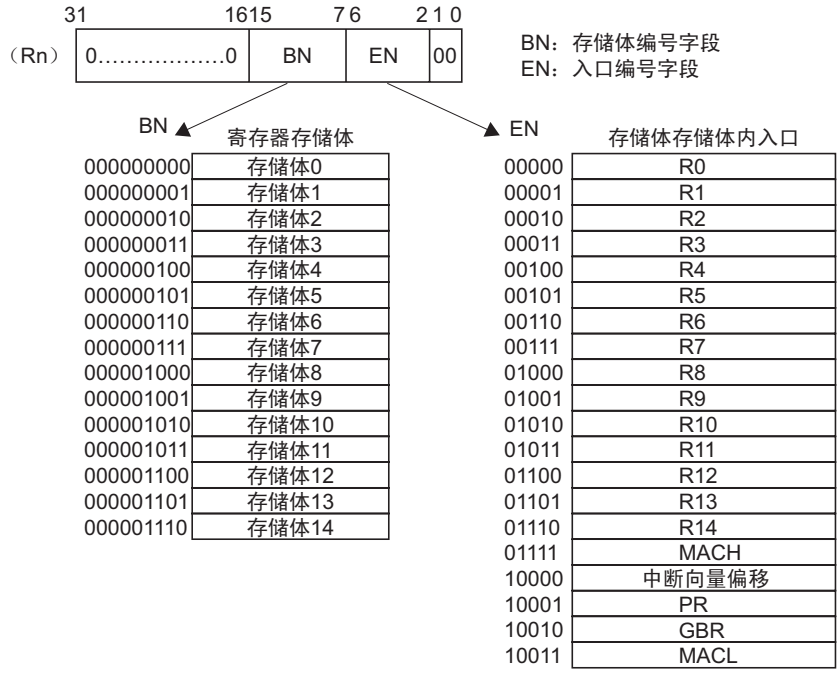
6.3.35 STBANK STore register BANK 系统控制指令

向指定存储体入口的寄存器保存 SH-2A/SH2A-FPU （新）

格式	操作概略	操作码	执行状态	T 位
STBANK R0, @Rn	R0 → （指定寄存器存储体入口）	0100nnnn11100001	7	—

(1) 说明

将 R0 传送到通用寄存器 Rn 的内容所指向的寄存器存储体中的 1 个入口。通用寄存器 Rn 指定寄存器存储体的编码与保存到存储体内的寄存器。



(2) 注意

在体系结构上，最多可持有 512 个存储体。
但是，存储体的数量因产品而异。

(3) 操作内容

```
STBANK ( long n)  /*STBANK  R0, @Rn */
  {
   Write_Bank_Long (R[n], R[0])
 PC+=2;
}
```

(4) 使用例

```
STBANK R0,@R1        ;执行前  R1=H'00000108, R0=H'FFFFFFFF
                     ;执行后  保存到存储体 2 的 R2=H'FFFFFFFF
```

6.3.36 STC STore Control register 系统控制指令

从控制寄存器存储 SH-2A/SH2A-FPU （新）

格式	操作概略	操作码	执行状态	T 位
STC TBR, Rn	TBR→Rn	0000nnnn01001010	1	—

(1) 说明

将控制寄存器 TBR 的数据保存到目的地。

(2) 操作内容

```
STCTBR(long n)  /* STC TBR, Rn*/
{
  R[n]=TBR;
  PC+=2;
}
```

(3) 使用例

```
STC TBR, R0                    ;执行前  R0=H'12345678, TBR=H'00000000
                               ;执行后  R0=H'00000000
```

6.4 SH-2E 的 CPU 指令

6.4.1 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
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 的内容加上 Rm 的内容，结果保存到 Rn。
也可以是通用寄存器 Rn 的内容加上 8 位立即数。
因为 8 位立即数符号扩展为 32 位，所以也可以进行减法运算。

(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
```

6.4.2 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
ADDC Rm,Rn	Rn+Rm+T→Rn, 进位 →T	0011nnnnmmmm1110	1	进位

(1) 说明

通用寄存器 Rn 的内容和 Rm 的内容、 T 位相加，结果保存到 Rn。根据运算结果将进位反映到 T 位，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
```

6.4.3 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
ADDV Rm,Rn	Rn+Rm→Rn, 上溢 →T	0011nnnnmmmm1111	1	上溢

(1) 说明

通用寄存器 Rn 的内容加上 Rm 的内容，结果保存到 Rn。发生上溢时， T 位被置位。

(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
```

6.4.4 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
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 的内容和 Rm 的逻辑与，结果保存到 Rn。

也可以是通用寄存器 R0 和 8 位立即数 （零扩展后）的逻辑与，或是 8 位存储器 （带变址 GBR 间接寻址方式下）和 8 位立即数的逻辑与。

(2) 注意

AND #imm,R0 的运算结果是 R0 的高 24 位总是被清除。

(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
```

6.4.5 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
BF label	T=0 时， disp×2+PC→PC, T=1 时， nop	10001011dddddddd	3/1	—

(1) 说明

是参照 T 位的带条件的转移指令。当 T=1 时，执行下一条指令；当 T=0 时，进行转移。

转移地址为 PC 加上位移量后的地址。PC 是本指令的 4 个地址后的起始地址。因为 8 位位移量在进行符号扩展后乘 2，所以和转移地址的相对距离为 –256 字节～ +254 字节。如果达不到转移地址，就需要通过和 BRA 指令等组合来解决。

(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) 使用例

```
CLRT                            ; 总是 T=0
BT    TRGET_T                   ; 因为 T=0，所以不发生转移。
BF    TRGET_F                   ; 因为 T=0，所以转移到 TRGET_F。
NOP                             ;
NOP                             ;← 用于计算 BF 指令转移地址的 PC 位置
TRGET_F:                        ;←BF 指令的转移地址
```

6.4.6 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
BF/S label	T=0 时 disp×2+PC→PC, T=1 时 nop	10001111dddddddd	2/1	—

(1) 说明

是参照 T 位的带条件的延迟转移指令。当 T=1 时，执行下一条指令；当 T=0 时，在执行完下一条指令后进行转移。

转移地址是 PC 加上位移量后的地址。PC 是本指令的 4 个地址后的起始地址。因为 8 位位移量在进行符号扩展后乘 2，所以和转移地址的相对距离为 –256 字节～ +254 字节。如果达不到转移地址，就需要通过和 BRA 指令等组合来解决。

(2) 注意

因为是延迟转移指令，所以先执行本指令后的下一条指令，然后进行转移。

在本指令和下一条指令之间不接受地址错误和中断。如果下一条指令是转移指令，就视为槽非法指令。

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) 使用例

```
        CLRT                    ; 总是 T=0
        BT/S TRGET_T            ; 因为 T=0，所以不发生转移。
        NOP                     ;
        BF/S TRGET_F            ; 因为 T=0，所以转移到 TRGET。
        ADD R0,R1               ; 在转移前执行。
        NOP                     ;← 用于计算 BF/S 指令转移地址的 PC 位置
    TRGET_F:                    ;←BF/S 指令的转移地址
```

【注】 在延迟转移中，转移操作发生在执行槽指令之后，但还必须按照延迟转移指令 → 延迟槽指令的顺序执行指令 （寄存器更新等）。例如：既使在延迟槽更改保存转移地址的寄存器，但更改前的寄存器内容仍为转移地址。

6.4.7 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
BRA label	disp×2+PC→PC	1010dddddddddddd	2	—

(1) 说明

是无条件延迟转移指令。转移地址为 PC 加上位移量的地址。PC 为本指令的 4 个地址后的起始地址。因为 12 位位移量在进行符号扩展后乘 2，所以和转移地址的相对距离为 –4096 字节～ +4094 字节。如果达不到转移地址，就需要在用 MOV 指令将转移地址传送到寄存器后，更改为 JMP 指令。

(2) 注意

因为是延迟转移指令，所以先执行本指令后的下一条指令，然后进行转移。

在本指令和下一条的指令之间不接受地址错误和中断。如果下一条指令是转移指令，就视为槽非法指令。

(3) 操作内容

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

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

(4) 使用例

```
BRA   TRGET                ; 转移到 TRGET。
ADD   R0,R1                ; 在转移前执行。
NOP                        ;← 用于计算 BRA 指令转移地址的 PC 位置
TRGET:                     ;←BRA 指令的转移地址
```

【注】 在延迟转移中，转移操作发生在执行槽指令之后，但还必须按照延迟转移指令 → 延迟槽指令的顺序执行指令 （寄存器更新等）。例如：即使在延迟槽更改保存转移地址的寄存器，但更改前的寄存器内容仍为转移地址。

6.4.8 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
BRAF Rm	Rm+PC→PC	0000mmmm00100011	2	—

(1) 说明

这是无条件延迟转移指令。转移地址为 PC 加上通用寄存器 Rm 的内容的 32 位后的地址。 PC 是本指令的 4 个地址后的起始地址。

(2) 注意

因为是延迟转移指令，所以先执行紧接着本指令之后的指令，然后进行转移。

在本指令和紧接着的指令之间不接受地址错误和中断。如果紧接着的指令为转移指令，就视为槽非法指令。

(3) 操作内容

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

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

(4) 使用例

```
    MOV.L #(TRGET-BRAF_PC),R0     ; 设定位移量。
    BRAF  R0                      ; 转移到 TRGET。
    ADD   R0,R1                   ; 在转移前执行。
BRAF_PC:                          ;← 用于计算 BRAF 指令转移地址的 PC 位置
    NOP
TRGET:                            ;←BRAF 指令的转移地址
```

【注】 在延迟转移中，转移操作发生在执行槽指令之后，但还必须按照延迟转移指令 → 延迟槽指令的顺序执行指令 （寄存器更新等）。例如：即使在延迟槽更改保存转移地址的寄存器，但更改前的寄存器内容仍为转移地址。

6.4.9 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
BSR label	PC→PR, disp×2+PC→PC	1011ddddddddddddd	2	—

(1) 说明

延迟转移到指定地址的子程序过程。在将 PC 的内容保存到 PR 后，转移到 PC 加上位移量后的地址。 PC 为本指令的 4 个地址后的起始地址。因为 12 位位移量在进行符号扩展后乘 2，所以和转移地址的相对距离为 –4096 字节～ +4094 字节。如果达不到转移地址，就需要在用 MOV 指令将转移地址传送到寄存器后，更改为 JSR 指令。和 RTS 组合用于子程序过程的调用。

(2) 注意

因为是延迟转移指令，所以先执行紧接着本指令之后的指令，然后进行转移。

在本指令和紧接着的指令之间不接受地址错误和中断。如果紧接着的指令为转移指令，就视为槽非法指令。

(3) 操作内容

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

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

(4) 使用例

```
BSR   TRGET                ; 转移到 TRGET
MOV   R3,R4                ; 在转移前执行。
ADD   R0,R1                ;← 用于计算 BSR 指令转移地址的 PC 位置
・・・・・                  这是从过程返回的地址 （PR 的内容）。
・・・・・
TRGET:                     ;← 过程的入口
MOV   R2,R3                ;
RTS                        ; 返回到上述的 ADD 指令。
MOV   #1,R0                ; 在转移前执行。
```

【注】 在延迟转移中，转移的操作是发生在执行槽指令后，但是还必须按照延迟转移指令 → 延迟槽指令的顺序执行指令（寄存器更新等）。例如：即使在延迟槽更改保存转移地址的寄存器，但更改前的寄存器内容仍为转移地址。

6.4.10 BSRF Branch to SubRoutine Far：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
BSRF Rm	PC→PR, Rm+PC→PC	0000mmmm00000011	2	—

(1) 说明

延迟转移到指定地址的子程序过程。将 PC 的内容保存到 PR。转移地址为 PC 加上通用寄存器 Rm 内容的 32 位数据后的地址。 PC 为本指令的 4 个地址后的起始地址。和 RTS 组合用于子程序过程的调用。

(2) 注意

因为是延迟转移指令，所以先执行紧接着本指令之后的指令，然后进行转移。

在本指令和紧接着的指令之间不接受地址错误和中断。如果紧接着的指令为转移指令，就视为槽非法指令。

(3) 操作内容

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

(4) 使用例

```
MOV.L #(TRGET-BSRF_PC),R0     ; 设定位移量。
          BSRF   R0           ; 转移到 TRGET。
          MOV    R3,R4        ; 在转移前执行。
       BSRF_PC:                ;← 用于计算 BSRF 指令转移地址的 PC 位置
          ADD    R0,R1        ;
          • • • • •
          • • • • •
       TRGET:                  ;← 过程的入口
           MOV   R2,R3        ;
           RTS                ; 返回到上述的 ADD 指令。
           MOV   #1,R0        ; 在转移前执行。
```

【注】 在延迟转移中，转移操作发生在执行槽指令之后，但还必须按照延迟转移指令 → 延迟槽指令的顺序执行指令 （寄存器更新等）。例如：即使在延迟槽更改保存转移地址的寄存器，但更改前的寄存器内容仍为转移地址。

6.4.11 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
BT label	T=1 时 disp×2+PC→PC, T=0 时 nop	10001001dddddddd	3/1	—

(1) 说明

是参照 T 位的带条件的转移指令。当 T=1 时，进行转移；当 T=0 时，执行下一条指令。

转移地址为 PC 加上位移量后的地址。PC 为本指令的 4 个地址后的起始地址。因为 8 位位移量在进行符号扩展后乘 2，所以和转移地址的相对距离为 –256 字节～ +254 字节。如果达不到转移地址，就需要通过和 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

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

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

(4) 使用例

```
    SETT                    ;T=1
    BF    TRGET_F           ;T=1，所以不发生转移。
    BT    TRGET_T           ;T=1，所以转移到 TRGET_T。
    NOP                     ;
    NOP                     ;← 用于计算 BT 指令转移地址的 PC 位置
TRGET_T:                    ;←BT 指令的转移地址
```

6.4.12 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
BT/S label	T=1 时 disp×2+PC→PC, T=0 时 nop	10001101dddddddd	2/1	—

(1) 说明

是参照 T 位的带条件的延迟转移指令。当 T=1 时，在执行下一条指令后进行转移；当 T=0 时，执行下一条指令。

转移地址为 PC 加上位移量后的地址。PC 为本指令的 4 个地址后的起始地址。因为 8 位位移量在进行符号扩展后乘 2，所以和转移地址的相对距离为 –256 字节～ +254 字节。如果达不到转移地址，就需要通过和 BRA 指令的组合来解决。

(2) 注意

因为是延迟转移指令，所以先执行紧接着本指令之后的指令，然后进行转移。

在本指令和紧接着的指令之间不接受地址错误和中断。如果紧接着的指令为转移指令，就视为槽非法指令。

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) 使用例

```
        SETT                    ;T=1
        BF/S TRGET_F            ;T=1，所以不发生转移。
        NOP                     ;
        BT/S TRGET_T            ;T=1，所以转移到 TRGET_T。
        ADD R0,R1               ; 在转移前执行。
        NOP                     ;← 用于计算 BT/S 指令转移地址的 PC 位置
TRGET_T:                        ;←BT/S 指令的转移地址
```

【注】 在延迟转移中，转移操作发生在执行槽指令之后，但还必须按照延迟转移指令 → 延迟槽指令的顺序执行指令 （寄存器更新等）。例如：即使在延迟槽更改保存转移地址的寄存器，但更改前的寄存器内容仍为转移地址。

6.4.13 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
CLRMAC	0→MACH, MACL	0000000000101000	1	—

(1) 说明

清除 MACH 和 MACL 寄存器。

(2) 操作内容

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

(3) 使用例

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

6.4.14 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
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) 使用例

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

6.4.15 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
CMP/EQ Rm,Rn	Rn=Rm 时 1→T	0011nnnnmmmm0000	1	比较结果
CMP/GE Rm,Rn	带符号 Rn ≥ Rm 时 1→T	0011nnnnmmmm0011	1	比较结果
CMP/GT Rm,Rn	带符号 Rn > Rm 时 1→T	0011nnnnmmmm0111	1	比较结果
CMP/HI Rm,Rn	无符号 Rn>Rm 时 1→T	0011nnnnmmmm0110	1	比较结果
CMP/HS Rm,Rn	无符号 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	任意的字节相等时 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 和 Rm 的内容，如果指定的条件 (cond) 成立，就将 T 位置位；如果条件不成立，就清除 T 位。 Rn 的内容不变，能指定 8 个条件。 PZ 和 PL 的 2 个条件为 Rn 和 0 的比较。

EQ 的条件也可以用于 8 位立即数 （符号扩展后）和 R0 的比较。 R0 的内容不变。

助记符	说明
CMP/EQ Rm,Rn	Rn=Rm 时 T=1
CMP/GE Rm,Rn	带符号 Rn ≥ Rm 时 T=1
CMP/GT Rm,Rn	带符号 Rn > Rm 时 T=1
CMP/HI Rm,Rn	无符号 Rn > Rm 时 T=1
CMP/HS Rm,Rn	无符号 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	任意的字节相等时 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>>24)&0x000000FF;
        HL=(temp>>16)&0x000000FF;
        LH=(temp>>8)&0x000000FF;
        LL=temp&0x000000FF;
        HH=HH&&HL&&LH&&LL;
        if (HH==0) T=1;
        else T=0;
        PC+=2;
    }
    CMPIM(long i)    /* CMP_EQ #imm,R0 */
    {
        long imm;

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

(3) 使用例

```
    CMP/GE  R0,R1               ;R0=H'7FFFFFFF,R1=H'80000000
        BT      TRGET_T         ;T=0, 所以不发生转移。
        CMP/HS  R0,R1           ;R0=H'7FFFFFFF,R1=H'80000000
        BT      TRGET_T         ;T=1, 所以进行转移。
        CMP/STR R2,R3           ;R2="ABCD",R3="XYCZ"
        BT      TRGET_T         ;T=1, 所以进行转移。
```

6.4.16 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
DIV0S Rm,Rn	Rn 的 MSB→Q, Rm 的 MSB→M, M ^ Q→T	0010nnnnmmmm0111	1	计算结果

(1) 说明

对带符号除法进行初始设定。和本指令之后的 DIV1 （1 位除法运算）等指令组合，反复进行除法运算以求商。详细内容请参照 DIV1 的说明。

(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) 使用例

请参照 DIV1 的使用例。

6.4.17 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
DIV0U	0→M/Q/T	0000000000011001	1	0

(1) 说明

对无符号除法运算进行初始设定。和本指令之后的 DIV1 （1 位除法运算）等指令组合，反复进行除法运算以求商。详细内容请参照 DIV1 的说明。

(2) 操作内容

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

(3) 使用例

请参照 DIV1 的使用例。

6.4.18 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
DIV1 Rm,Rn	单步除法 (Rn÷Rm)	0011nnnnmmmm0100	1	计算结果

(1) 说明

是执行 1 位除法运算 （单步除法）的指令，通用寄存器 Rn 的 32 位内容 （被除数）除 Rm 的内容 （除数）。通过单独执行本指令或者和其他指令组合反复执行来求商。在反复执行时，不能改写所指定的寄存器和 M、 Q、 T 位。

所谓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。要通过除法求余数时，必须在使用 DIV1 指令求商后用下式计算：

（被除数） – （除数）× （商）= （余数）

另外，搭载了作为外围功能的除法器的 SH7600 系列中，能使用除法器的功能求余数。

没有检测被零除和上溢的功能，所以必须在除法运算前检查被零除和上溢。也没有求余数的运算功能，所以必须用被除数减去除数和求得的商的积后求余数。

首先通过 DIV0S 或者 DIV0U 进行初始设定，然后反复执行除数的位数次 DIV1。如果需要不低于 17 位的商，就将 ROTCL 置于 DIV1 之前。详细的除法顺序请参照下述使用例。

(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) 使用例 1

```
                                ;R1(32 位 )÷R0(16 位 )=R1(16 位 )：无符号
        SHLL16    R0            ; 将除数设定到高 16 位，低 16 位设定为 0
        TST       R0,R0         ; 被零除的检查
        BT        ZERO_DIV      ;
        CMP/HS    R0,R1         ; 上溢的检查
        BT        OVER_DIV      ;
        DIV0U                   ; 标志的初始化
        .arepeat  16            ;
        DIV1      R0,R1         ; 反复 16 次
        .aendr                  ;
        ROTCL     R1            ;
        EXTU.W    R1,R1         ;R1= 商
```

(4) 使用例 2

```
                                ;R1:R2(64 位 )÷R0(32 位 )=R2(32 位 )：无符号
        TST       R0,R0         ; 被零除的检查
        BT        ZERO_DIV      ;
        CMP/HS    R0,R1         ; 上溢的检查
        BT        OVER_DIV      ;
        DIV0U                   ; 标志的初始化
        .arepeat  32            ;
        ROTCL     R2            ; 反复 32 次
        DIV1      R0,R1         ;
        .aendr                  ;
        ROTCL     R2            ;R2= 商
```

(5) 使用例 3

```
                                ;R1(16 位 )÷R0(16 位 )=R1(16 位 )：带符号
        SHLL16    R0            ; 将除数设定到高 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         ; 当被除数是负时，减 1。
        DIV0S     R0,R1         ; 标志的初始化
        .arepeat   16           ;
        DIV1      R0,R1         ; 反复 16 次
        .aendr                  ;
        EXTS.W    R1,R1         ;
        ROTCL     R1            ;R1= 商 （1 的补码）
        ADDC      R2,R1         ; 当商的 MSB 是 1 时，加 1 后转换为 2 的补码
        EXTS.W    R1,R1         ;R1= 商 （2 的补码）
```

(6) 使用例 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         ; 当被除数是负时，减 1 后转换为 1 的补码
        DIV0S     R0,R1         ; 标志的初始化
        .arepeat   32           ;
        ROTCL     R2            ; 反复 32 次
        DIV1      R0,R1         ;
        .aendr                  ;
        ROTCL     R2            ;R2= 商 （1 的补码）
        ADDC      R3,R2         ; 当商的 MSB 是 1 时，加 1 后转换为 2 的补码
                                ;R2= 商 （2 的补码）
```

6.4.19 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
DMULS.L Rm,Rn	带符号 Rn×Rm→MACH, MACL	0011nnnnmmmm1101	2	—

(1) 说明

对通用寄存器 Rn 的内容和 Rm 的进行 32 位乘法运算， 64 位的结果保存到 MACH 寄存器和 MACL 寄存器。运算为带符号的算数运算。

(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) 使用例

```
DMULS  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）
```

6.4.20 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
DMULU.L Rm,Rn	无符号 Rn×Rm→MACH, MACL	0011nnnnmmmm0101	2	—

(1) 说明

对通用寄存器 Rn 的内容和 Rm 进行 32 位乘法运算， 64 位的结果保存到 MACH 寄存器和 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) 使用例

```
DMULU  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）
```

6.4.21 DT Decrement and Test：算术运算指令

递减和测试

格式	操作概略	操作码	执行状态	T 位
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 的内容减 1，结果和 0 （零）进行比较。当结果为 0 时，将 T 位置 1；否则，将 T 位置 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) 使用例

```
   MOV  #4,R5              ; 设定循环次数。
LOOP:
   ADD  R0,R1              ;
   DT R5                   ; 将 R5 的值进行递减，判断是否为 0。
   BF LOOP                 ; 如果 T=0，就转移到 LOOP （在此例中循环 4 次）
```

6.4.22 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
EXTS.B Rm,Rn	Rm 从字节符号扩展 →Rn	0110nnnnmmmm1110	1	—
EXTS.W Rm,Rn	Rm 从字符号扩展 →Rn	0110nnnnmmmm1111	1	—

(1) 说明

将通用寄存器 Rm 的内容进行符号扩展，结果保存到 Rn。

当指定字节时，将 Rm 的 bit7 的内容传送到 Rn 的 bit8 ～ bit31；当指定字时，将 Rm 的 bit15 的内容传送到 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
```

6.4.23 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
EXTU.B Rm,Rn	Rm 从字节零扩展 →Rn	0110nnnnmmmm1100	1	—
EXTU.W Rm,Rn	Rm 从字零扩展 →Rn	0110nnnnmmmm1101	1	—

(1) 说明

将通用寄存器 Rm 的内容进行零扩展，结果保存到 Rn。
当指定字节时，将 0 传送到 Rn 的 bit8 ～ bit31；当指定字时，将 0 传送到 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
```

6.4.24 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
JMP @Rm	Rm→PC	0100mmmm00101011	2	—

(1) 说明

无条件地延迟转移到寄存器间接指定的地址。转移地址是通用寄存器 Rm 内容的 32 位数据所表示的地址。

(2) 注意

因为是延迟转移指令，所以先执行紧接着本指令之后的指令，然后进行转移。

在本指令和紧接着的指令之间不接受地址错误和中断。如果紧接着的指令是转移指令，就视为槽非法指令。

(3) 操作内容

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

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

(4) 使用例

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

【注】 在延迟转移中，转移操作发生在执行槽指令之后，但还必须按照延迟转移指令 → 延迟槽指令的顺序执行指令（寄存器更新等）。例如：即使在延迟槽更改保存转移地址的寄存器，但更改前的寄存器内容仍为转移地址。

6.4.25 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
JSR @Rm	PC→PR, Rm→PC	0100mmmm00001011	2	—

(1) 说明

延迟转移到寄存器间接指定地址的子程序过程。将 PC 的内容保存到 PR 后，转移到以通用寄存器 Rm 内容的 32 位所表示的地址。被保存的 PC 为本指令的 4 个地址后的起始地址。和 RTS 组合用于子程序过程的调用。

(2) 注意

因为是延迟转移指令，所以先执行紧接着本指令之后的指令，然后进行转移。

在本指令和紧接着的指令之间不接受地址错误和中断。如果紧接着的指令是转移指令，就视为槽非法指令。

(3) 操作内容

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

(4) 使用例

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

【注】 在延迟转移中，转移操作发生在执行槽指令之后，但还必须按照延迟转移指令 → 延迟槽指令的顺序执行指令 （寄存器更新等）。例如：即使在延迟槽更改保存转移地址的寄存器，但更改前的寄存器内容仍为转移地址。

6.4.26 LDC LoaD to Control register：系统控制指令

向控制寄存器的加载

格式		操作概略	操作码	执行状态	T 位
LDC	Rm,SR	Rm→SR	0100mmmm00001110	3	LSB
LDC	Rm,GBR	Rm→GBR	0100mmmm00011110	1	—
LDC	Rm,VBR	Rm→VBR	0100mmmm00101110	1	—
LDC.L	@Rm+,SR	(Rm)→SR, Rm+4→Rm	0100mmmm00000111	5	LSB
LDC.L	@Rm+,GBR	(Rm)→GBR, Rm+4→Rm	0100mmmm00010111	1	—
LDC.L	@Rm+,VBR	(Rm)→VBR, Rm+4→Rm	0100mmmm00100111	1	—

(1) 说明

将源操作数保存到控制寄存器 SR、 GBR 和 VBR。

(2) 操作内容

```
LDCSR(long m)    /* LDC Rm,SR */
{
    SR=R[m]&0x000063F3;
    PC+=2;
}

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

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


LDCMSR(long m)    /* LDC.L @Rm+,SR */
{
    SR=Read_Long(R[m])&0x000063F3;
    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;
    }
```

(3) 使用例

```
LDC     R0,SR                   ;执行前 R0=H'FFFFFFFF,SR=H'00000000
                                ;执行后 SR=H'000063F3
LDC.L   @R15+,GBR               ;执行前 R15=H'10000000
                                ;执行后 R15=H'10000004,GBR=@H'10000000
```

6.4.27 LDS LoaD to System register：系统控制指令

向系统寄存器的加载

格式	操作概略	操作码	执行状态	T 位
LDS Rm,MACH	Rm→MACH	0100mmmm00001010	1	—
LDS Rm,MACL	Rm→MACL	0100mmmm00011010	1	—
LDS Rm,PR	Rm→PR	0100mmmm00101010	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	—

(1) 说明

将源操作数保存到系统寄存器 MACH、 MACL 和 PR。

(2) 操作内容

```
LDSMACH(long m) /* LDS Rm,MACH */
{
    MACH=R[m];
    PC+=2;
}

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

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


LDSMMACH(long m) /* LDS.L @Rm+,MACH */
{
    MACH=Read_Long(R[m]);
    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;
}
```

(3) 使用例

```
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
```

6.4.28 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
MAC.L @Rm+,@Rn+	带符号 (Rn)×(Rm)+MAC→MAC	0000nnnnmmmm1111	4	—

(1) 说明

对将通用寄存器 Rm 和 Rn 的内容作为地址的 32 位操作数进行带符号的乘法运算，并将结果的 64 位加上 MAC 寄存器的内容后保存到 MAC 寄存器。每读一次操作数， Rm 和 Rn 都分别加 4。

当 S 位是 0 时，结果的 64 位保存到连接着的 MACH 和 MACL 寄存器。

当 S 位是 1 时，和 MAC 寄存器的加法运算为从 LSB 到第 48 位的饱和运算。在饱和运算中，只有 MAC 寄存器的低 48 位有效，结果被限制在 H'FFFF800000000000 （最小值）～ H'00007FFFFFFFFFFF （最大值）的范围内。

(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(fnLmL<0){
            Res2=~Res2;
            if (Res0==0) Res2++;
            else Res0=(~Res0)+1;
        }
        if(S==1){
            Res0=MACL+Res0;
            if (MACL>Res0) Res2++;
            if (MACH&0x00008000);
            else   Res2+=MACH?0xFFFF0000;
                   Res2+=MACH&0x00007FFF;


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

            MACH=(Res2&0x0000FFFF) (MACH&0xFFFF0000);
            MACL=Res0;
        }
        else {
            Res0=MACL+Res0;
            if (MACL>Res0) Res2++;
            Res2+=MACH;

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

(3) 使用例

```
        MOVA    TBLM,R0         ; 获得表的地址
        MOV     R0,R1           ;
        MOVA    TBLN,R0         ; 获得表的地址
        CLRMAC                  ;MAC 寄存器的初始化
        MAC.L   @R0+,@R1+       ;
        MAC.L   @R0+,@R1+       ;
        STS     MACL,R0         ;RO 获得结果
    . . . . . . . . . . . .
        .align  2               ;
TBLM                            .data.l  H'1234ABCD     ;
                                .data.l  H'5678EF01     ;
TBLN                            .data.l  H'0123ABCD     ;
                                .data.l  H'4567DEF0     ;
```

6.4.29 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
MAC.W @Rm+,@Rn+ MAC @Rm+,@Rn+	带符号 (Rn)×(Rm)+MAC→MAC	0100nnnnmmmm1111	3	—

(1) 说明

对将通用寄存器 Rm 和 Rn 的内容作为地址的 16 位操作数进行带符号的乘法运算，并将结果的 32 位加上 MAC 寄存器的内容后保存到 MAC 寄存器。每读一次操作数， Rm 和 Rn 都分别加 2。

当 S 位是 0 时，为 16×16+64→64 位的乘法累加运算，结果的 64 位保存到连接着的 MACH 和 MACL 寄存器。

当 S 位是 1 时，为 16×16+32→32 位的乘法累加运算，和 MAC 寄存器的加法运算为饱和运算。在饱和运算中，只有 MACL 寄存器有效，结果被限制在 H'80000000 （最小值）～ H'7FFFFFFF （最大值）的范围内。如果发生上溢，就将 MACH 寄存器设置为 H'00000001。如果向负方向上溢，就将 H'80000000 （最小值）保存在 MACL 寄存器；如果向正方向上溢，就将 H'7FFFFFFF （最大值）保存在 MACL 寄存器。

(2) 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) {
                MACH=0x00000001;
                if (src==0) MACL=0x7FFFFFFF;
                if (src==2) MACL=0x80000000;
        }
        }
        else {
             MACH+=tempn;
             if (templ>MACL) MACH+=1;
        }
        PC+=2;
    }
```

(3) 使用例

```
        MOVA    TBLM,R0         ; 获得表的地址
        MOV     R0,R1           ;
        MOVA    TBLN,R0         ; 获得表的地址
        CLRMAC                  ;MAC 寄存器的初始化
        MAC.W   @R0+,@R1+       ;
        MAC.W   @R0+,@R1+       ;
        STS     MACL,R0         ;RO 获得结果
      . . . . . . . . . . . .
        .align  2               ;
TBLM                            .data.w  H'1234;
                                .data.w  H'5678;
TBLN                            .data.w  H'0123;
                                .data.w  H'4567;
```

6.4.30 MOV MOVe data：数据传送指令

数据的传送

格式	操作概略	操作码	执行状态	T 位
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	0110nnnnmmmm0100	1	—
MOV.W @Rm+,Rn	→Rn,Rm+1→Rm	0110nnnnmmmm0101	1	—
MOV.L @Rm+,Rn	(Rm)→ 符号扩展	0110nnnnmmmm0110	1	—
MOV.B Rm,@(R0,Rn)	→Rn,Rm+2→Rm	0000nnnnmmmm0100	1	—
MOV.W Rm,@(R0,Rn)	(Rm)→Rn, Rm+4→Rm	0000nnnnmmmm0101	1	—
MOV.L Rm,@(R0,Rn)	Rm→(R0+Rn)	0000nnnnmmmm0110	1	—
MOV.B @(R0,Rm),Rn	Rm→(R0+Rn)	0000nnnnmmmm1100	1	—
MOV.W @(R0,Rm),Rn	Rm→(R0+Rn)	0000nnnnmmmm1101	1	—
MOV.L @(R0,Rm),Rn	(R0+Rm)→ 符号扩展 →Rn (R0+Rm)→ 符号扩展 →Rn (R0+Rm)→Rn	0000nnnnmmmm1110	1	—

(1) 说明

将源操作数传送到目的地。当操作数作为存储器时，能在字节、字或者长字范围内指定要传送的数据长度；当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到寄存器。

(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
```

6.4.31 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
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	—

(1) 说明

将符号扩展为长字的立即数保存到通用寄存器 Rn。当数据为字或者长字时，参照保存在 PC 加上位移量后的地址的表内数据。

当数据为字时， 8 位位移量在进行零扩展后乘 2，所以和表的相对距离是在 PC+510 字节的范围内。 PC 为本指令的 4 个地址后的起始地址。

当数据为长字时， 8 位位移量在进行零扩展后乘 4，所以和操作数的相对距离是在 PC+1020 字节的范围内。 PC 为本指令的 4 个地址后的起始地址，但是将 PC 的低 2 位校正为 B'00 后的值用于地址计算。

(2) 注意

此表最适于配置在模块后端或者无条件转移指令的 1 条指令之后。如果因为在 510 字节 /1020 字节内没有无条件转移指令等而不能进行优化配置，就需要使用 BRA 指令跳过该表。如果将本指令配置在紧接着延迟转移指令之后， PC 就为转移地址的 “起始地址 +2”。

瑞萨科技的 “SuperH RISC engine 汇编程序”中，必须使用被放大的值 （×2， ×4）表述位移量的值。

(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             ;← 用于计算 MOV.W 指令转移地址的 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 的转移地址
1014          CMP/EQ  #0,R0             ;← 用于计算 MOV.L 指令地址的 PC 位置
              .align  4                 ;
1018          .data.l H'12345678        ;
```

6.4.32 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
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

将源操作数传送到目的地数，最适于内部外围模块区内的数据存取。能在字节、字或者长字范围内指定数据长度，但是寄存器固定为 R0。给 GBR 设定内部外围模块的基址。

当内部外围模块的数据长度是字节时， 8 位位移量只进行零扩展，所以最大能指定到 +255 字节。当数据长度是字时， 8 位位移量在进行零扩展后乘 2，所以最大能指定到 +510 字节。当数据长度是长字时， 8 位位移量在进行零扩展后乘 4，所以最大能指定到 +1020 字节。如果达不到存储器操作数，就需要在将 GBR 传送到通用寄存器后，使用上述的 @(R0,Rn) 模式。

当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到寄存器。

(2) 注意

加载时的目标寄存器固定为 R0。因此，即使紧接着的指令要参照 R0，也必须等到加载指令执行结束为止。能通过改变指令的顺序进行优化处理。

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

瑞萨科技的 “SuperH RISC engine 汇编程序”中，必须使用被放大的值 （×1、 ×2、 ×4）表述位移量的值。

(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'FFFF7F80
```

6.4.33 MOV MOVe structure data：数据传送指令

比较结构体数据的传送

格式	操作概略	操作码	执行周期	T 位
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

将源操作数传送到目的地，最适于结构体和堆栈内的数据存取。能在字节、字或者长字范围内指定数据长度，但是当数据长度为字节或者字时，寄存器固定为 R0。

当数据长度是字节时， 4 位位移量只进行零扩展，所以最大能指定到 +15 字节。当数据长度是字时， 4 位位移量在进行零扩展后乘 2，所以最大能指定到 +30 字节。当数据长度是长字时， 4 位位移量在进行零扩展后乘 4，所以最大能指定到 +60 字节。如果达不到存储器操作数，就需要使用上述的 @(R0,Rn) 模式。

当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到寄存器。

(2) 注意

加载字节或者字数据时，目标寄存器固定为 R0。因此，即使紧接着的指令要参照 R0，也必须等到加载指令执行结束为止。能通过改变指令的顺序进行优化处理。

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

瑞萨科技的 “SuperH RISC engine 汇编程序”中，必须使用被放大的值 （×1、 ×2、 ×4）表述位移量的值。

(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
```

6.4.34 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
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到通用寄存器 R0。因为 8 位位移量在进行零扩展后乘 4，所以和操作数的相对距离是在 PC+1020 字节的范围内。 PC 为本指令的 4 个地址后的起始地址，但是将 PC 的低 2 位校正为 B'00 后的值用于计算地址。

(2) 注意

如果将本指令配置在紧接着延迟转移指令之后， PC 就为转移地址的 “起始地址 +2”。

瑞萨科技的 “SuperH RISC engine 汇编程序”中，必须使用被放大的值 （×4）表述位移量的值。

(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 位校正后的位置
100A      ADD     R4,R5         ;← 在 MOVA 指令计算地址时， PC 的原来位置
          .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                   ;
```

6.4.35 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
MOVT Rn	T→Rn	0000nnnn00101001	1	—

(1) 说明

将 T 位保存到通用寄存器 Rn。当 T=1 时， Rn=1 ； 当 T=0 时， Rn=0。

(2) 操作内容

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

(3) 使用例

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

6.4.36 MUL.L MULtiply Long：算术运算指令

双精度乘法

格式	操作概略	操作码	执行状态	T 位
MUL.L Rm,Rn	Rn×Rm→MACL	0000nnnnmmmm0111	2	—

(1) 说明

对通用寄存器 Rn 的内容和 Rm 进行 32 位乘法运算，结果的低 32 位保存到 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) 使用例

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

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

带符号乘法

格式	操作概略	操作码	执行状态	T 位
MULS.W Rm,Rn MULS Rm,Rn	带符号 Rn×Rm→MACL	0010nnnnmmmm1111	1	—

(1) 说明

对通用寄存器 Rn 的内容和 Rm 进行 16 位乘法运算，结果的 32 位保存到 MACL 寄存器。运算为带符号的算数运算。 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) 使用例

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

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

无符号乘法

格式	操作概略	操作码	执行状态	T 位
MULU.W Rm,Rn MULU Rm,Rn	无符号 Rn×Rm→MACL	0010nnnnmmmm1110	1	—

(1) 说明

对通用寄存器 Rn 的内容和 Rm 进行 16 位乘法运算，结果的 32 位保存到 MACL 寄存器。运算为无符号的算数运算。 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) 使用例

```
MULU  R0,R1                ;操作前 R0=H'00000002,R1=H'FFFFAAAA
                           ;操作后 MACL=H'00015554
STS   MACL,R0              ;获得运算结果
```

6.4.39 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
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，结果保存到 Rn。即， 0 减去 Rm，结果保存到 Rn。

(2) 操作内容

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

(3) 使用例

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

6.4.40 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
NEGC Rm,Rn	0-Rm-T→Rn, 借位 →T	0110nnnnmmmm1010	1	借位

(1) 说明

0 减去通用寄存器 Rm 的内容和 T 位，结果保存到 Rn。根据运算结果将借位反映到 T 位。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
```

6.4.41 NOP No Operation：系统控制指令

无操作

格式	操作概略	操作码	执行状态	T 位
NOP	无操作	0000000000001001	1	—

(1) 说明

只递增 PC，进入执行下一条指令。

(2) 操作内容

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

(3) 使用例

```
NOP                          ; 经过 1 个周期。
```

6.4.42 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
NOT Rm,Rn	~Rm→Rn	0110nnnnmmmm0111	1	—

(1) 说明

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

(2) 操作内容

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

(3) 使用例

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

6.4.43 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
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 的内容和 Rm 的逻辑或，结果保存到 Rn。

也可以是通用寄存器 R0 和 8 位立即数 （零扩展后）的逻辑或，或是 8 位存储器 （带变址的 GBR 间接寻址方式下）和 8 位立即数的逻辑或。

(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
```

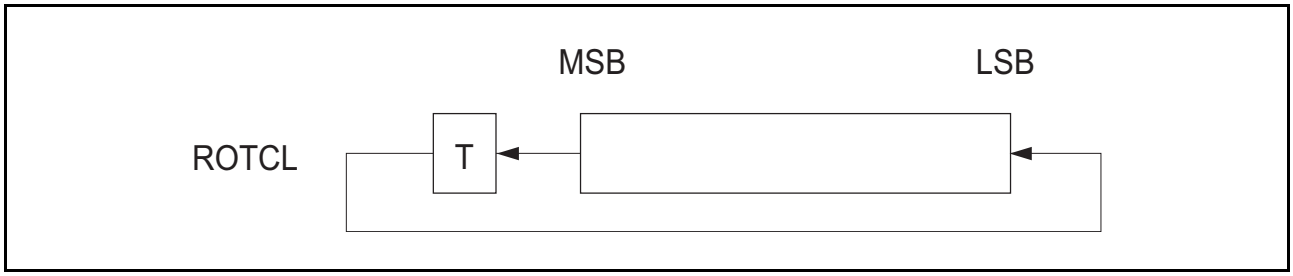
6.4.44 ROTCL ROTate with Carry Left：移位指令

带 T 位的左循环 1 位

格式	操作概略	操作码	执行状态	T 位
ROTCL Rn	T←Rn←T	0100nnnn00100100	1	MSB

(1) 说明

将通用寄存器 Rn 的内容带 T 位左循环 1 位，结果保存到 Rn。循环后的操作数的移出位传送到 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) 使用例

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

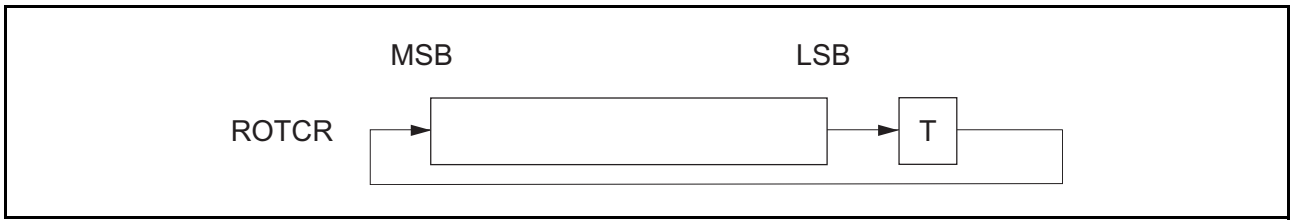
6.4.45 ROTCR ROTate with Carry Right：移位指令

带 T 位的右循环 1 位

格式	操作概略	操作码	执行状态	T 位
ROTCR Rn	T→Rn→T	0100nnnn00100101	1	LSB

(1) 说明

将通用寄存器 Rn 的内容带 T 位右循环 1 位，结果保存到 Rn。循环后的操作数的移出位传送到 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) 使用例

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

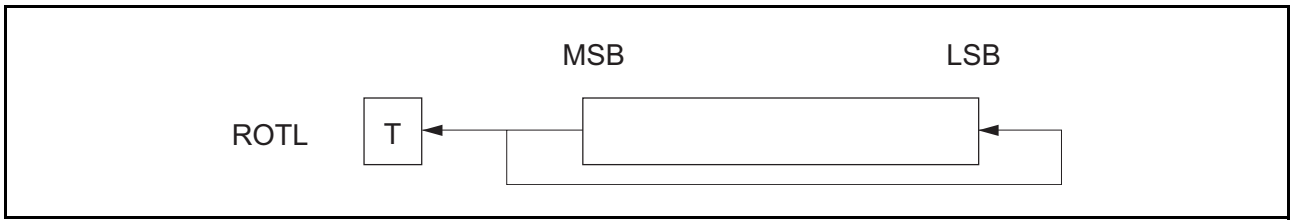
6.4.46 ROTL ROTate Left：移位指令

左循环 1 位

格式	操作概略	操作码	执行状态	T 位
ROTL Rn	T←Rn←MSB	0100nnnn00000100	1	MSB

(1) 说明

将通用寄存器 Rn 的内容左循环 1 位，结果保存到 Rn。循环后的操作数的移出位传送到 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;
    if (T==1) R[n]|=0x00000001;
    else R[n]&=0xFFFFFFFE;
    PC+=2;
}
```

(3) 使用例

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

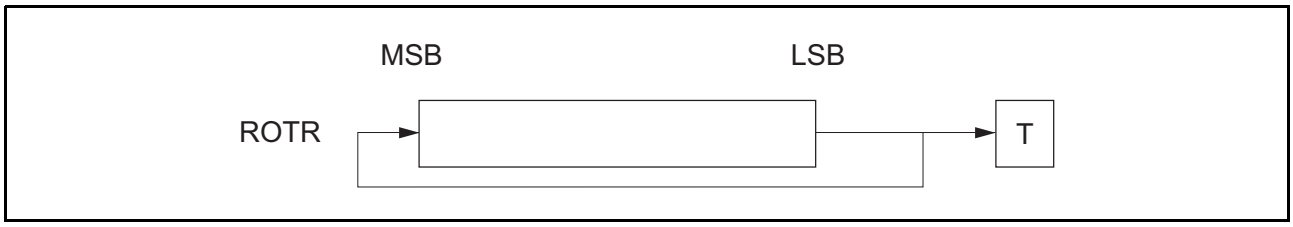
6.4.47 ROTR ROTate Right：移位指令

右循环 1 位

格式	操作概略	操作码	执行状态	T 位
ROTR Rn	LSB→Rn→T	0100nnnn00000101	1	LSB

(1) 说明

将通用寄存器 Rn 的内容右循环 1 位，结果保存到 Rn。循环后的操作数的移出位传送到 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) 使用例

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

6.4.48 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
RTE	堆栈区 →PC/SR	0000000000101011	6	—

(1) 说明

从中断程序返回。即，在从堆栈返回 PC 和 SR 后，从返回的 PC 所指示的地址开始继续处理。

(2) 注意

因为是延迟转移指令，所以先执行紧接着本指令之后的指令，然后进行转移。

在本指令和紧接着的指令之间不接受地址错误和中断。如果紧接着的指令是转移指令，就视为槽非法指令。

(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])&0x000063F3;
    R[15]+=4;
    Delay_Slot(temp+2);
}
```

(4) 使用例

```
RTE                         ; 返回到源程序。
ADD    #8,R14               ; 在转移前执行。
```

【注】 在延迟转移中，转移操作发生在执行槽指令之后，但还必须按照延迟转移指令 → 延迟槽指令的顺序执行指令 （寄存器更新等）。例如：即使在延迟槽更改保存转移地址的寄存器，但更改前的寄存器内容仍为转移地址。

6.4.49 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
RTS	PR→PC	0000000000001011	2	—

(1) 说明

从子程序过程返回。即，在从 PR 返回 PC 后，从返回的 PC 所指示的地址开始继续处理。能通过本指令从 BSR 和 JSR 指令调用的子程序过程返回到调用源。

(2) 注意

因为是延迟转移指令，所以先执行紧接着本指令之后的指令，然后进行转移。

在本指令和紧接着的指令之间不接受地址错误和中断。如果紧接着的指令是转移指令，就视为槽非法指令。

(3) 操作内容

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

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

(4) 使用例

```
        MOV.L       TABLE,R3        ;R3=TRGET 的地址
        JSR         @R3             ; 转移到 TRGET。
        NOP                         ; 在 JSR 前执行。
        ADD         R0,R1           ;← 从过程返回的地址 （PR 的内容）
        ・・・・・・・・・
TABLE:  .data.l     TRGET           ; 跳转表
        ・・・・・・・・・
TRGET:  MOV         R1,R0           ;← 过程的入口
        RTS                         ;PR 的内容 ->PC
        MOV         #12,R0          ; 在转移前执行。
```

【注】 在延迟转移中，转移操作发生在执行槽指令之后，但还必须按照延迟转移指令 → 延迟槽指令的顺序执行指令 （寄存器更新等）。例如：即使在延迟槽更改保存转移地址的寄存器，但更改前的寄存器内容仍为转移地址。

6.4.50 SETT SET Tbit：系统控制指令

T 位的置位

格式	操作概略	操作码	执行状态	T 位
SETT	1→T	0000000000011000	1	1

(1) 说明

将 T 位置位。

(2) 操作内容

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

(3) 使用例

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

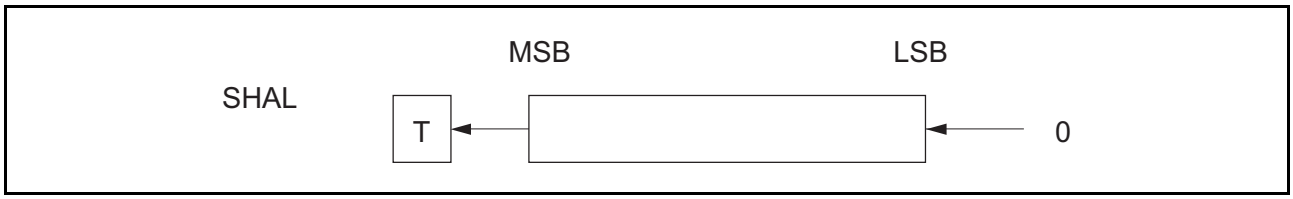
6.4.51 SHAL SHift Arithmetic Left：移位指令

算术左移 1 位

格式	操作概略	操作码	执行状态	T 位
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，结果保存到 Rn。移位后的操作数的移出位传送到 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) 使用例

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

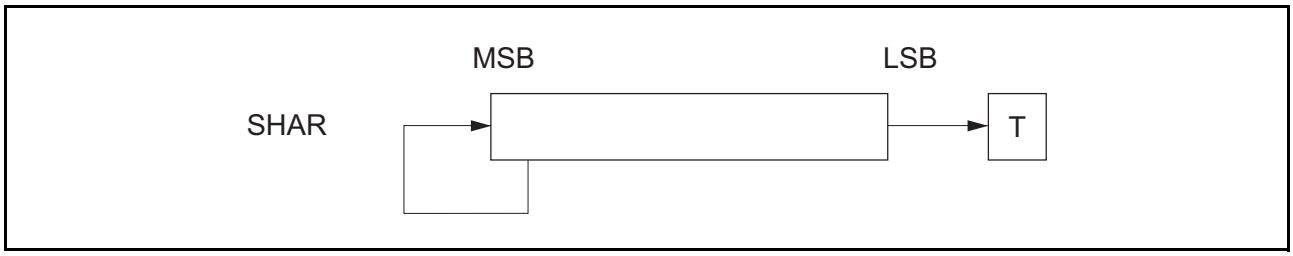
6.4.52 SHAR SHift Arithmetic Right：移位指令

算术右移 1 位

格式	操作概略	操作码	执行状态	T 位
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，结果保存到 Rn。移位后的操作数的移出位传送到 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) 使用例

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

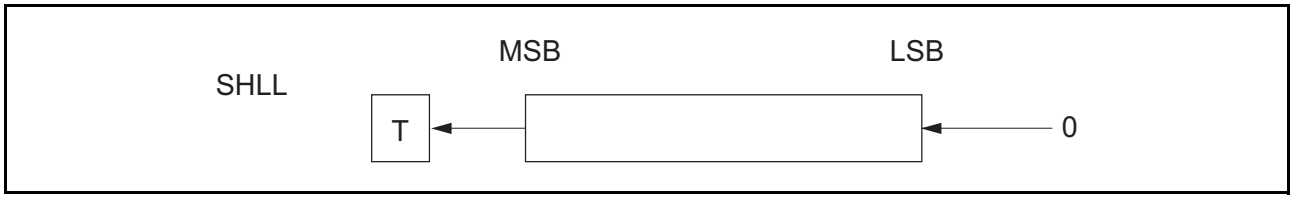
6.4.53 SHLL SHift Logical Left：移位指令

逻辑左移 1 位

格式	操作概略	操作码	执行状态	T 位
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，结果保存到 Rn。移位后的操作数的移出位传送到 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) 使用例

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

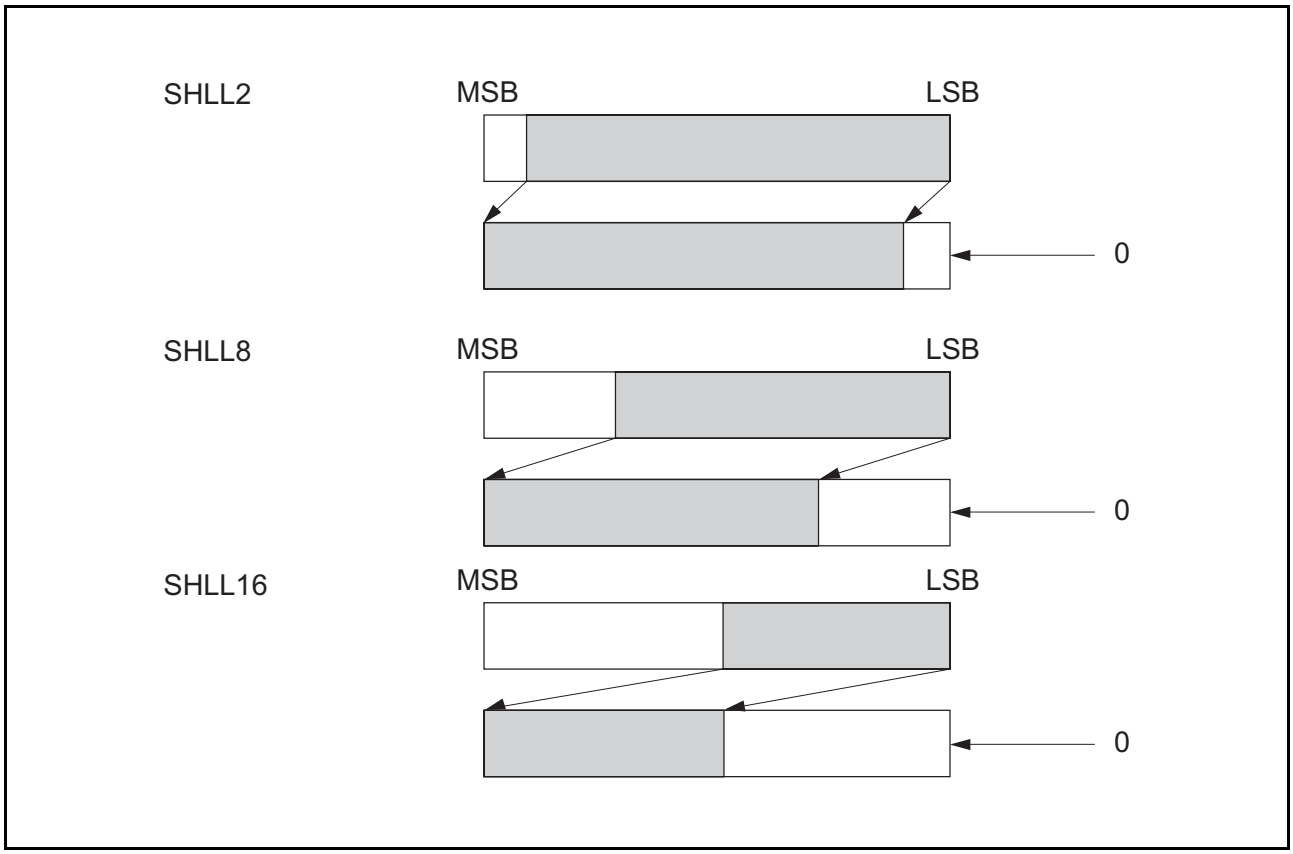
6.4.54 SHLLn n bits SHift Logical Left：移位指令

逻辑左移 n 位 左移

格式	操作概略	操作码	执行状态	T 位
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，结果保存到 Rn。舍去操作数的移出位。



(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
```

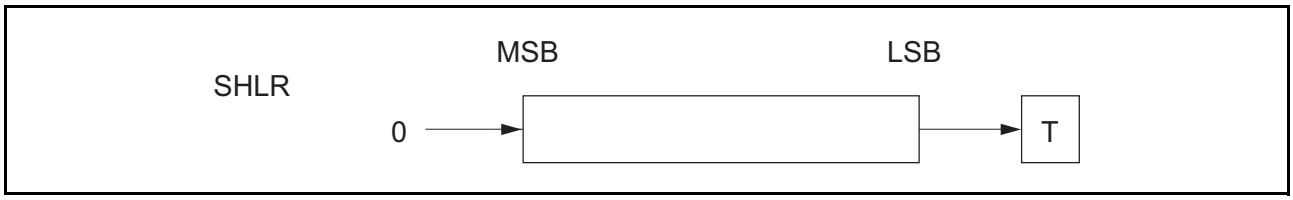
6.4.55 SHLR SHift Logical Right：移位指令

逻辑右移 1 位

格式	操作概略	操作码	执行状态	T 位
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，结果保存到 Rn。移位后的操作数的移出位传送到 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) 使用例

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

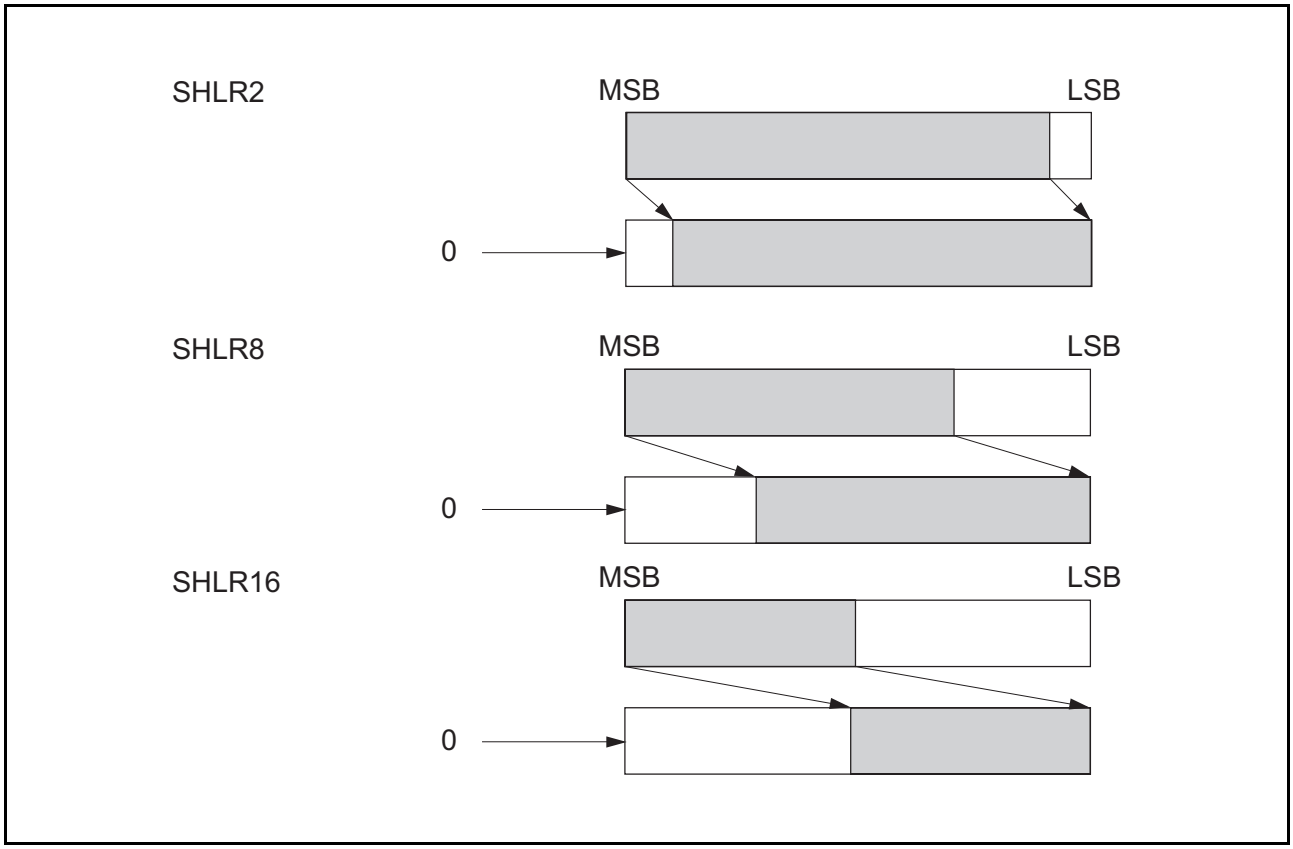
6.4.56 SHLRn n bits SHift Logical Right：移位指令

逻辑右移 n 位

格式	操作概略	操作码	执行状态	T 位
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，结果保存到 Rn。舍去操作数的移出位。



(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
```

6.4.57 SLEEP SLEEP：系统控制指令

向低功耗状态的转移

格式	操作概略	操作码	执行状态	T 位
SLEEP	睡眠	0000000000011011	5	—

(1) 说明

使 CPU 进入低功耗状态。

在低功耗状态下，保持 CPU 的内部状态，停止执行紧接着的指令，等待中断请求的产生。当产生中断请求时，就退出低功耗状态。

(2) 注意

执行状态的 5 是指转移到睡眠模式前的状态数。

(3) 操作内容

```
SLEEP( )    /* SLEEP */
{
    wait_for_exception;
}
```

(4) 使用例

```
SLEEP               ; 向低功耗状态的转移
```

6.4.58 STC STore Control register：系统控制指令

从控制寄存器存储

格式	操作概略	操作码	执行状态	T 位
STC SR,Rn	SR→Rn	0000nnnn00000010	2	—
STC GBR,Rn	GBR→Rn	0000nnnn00010010	1	—
STC VBR,Rn	VBR→Rn	0000nnnn00100010	1	—
STC.L SR,@-Rn	Rn-4→Rn, SR→(Rn)	0100nnnn00000011	2	—
STC.L GBR,@-Rn	Rn-4→Rn, GBR→(Rn)	0100nnnn00010011	1	—
STC.L VBR,@-Rn	Rn-4→Rn, VBR→(Rn)	0100nnnn00100011	1	—

(1) 说明

将控制寄存器 SR、 GBR 或者 VBR 保存到目的地。

(2) 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;
}

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;
}
```

(3) 使用例

```
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
```

6.4.59 STS STore System register：系统控制指令

从系统寄存器存储

格式		操作概略	操作码	执行状态	T 位
STS	MACH,Rn	MACH→Rn	0000nnnn00001010	1	—
STS	MACL,Rn	MACL→Rn	0000nnnn00011010	1	—
STS	PR,Rn	PR→Rn	0000nnnn00101010	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	—

(1) 说明

将系统寄存器 MACH、 MACL 或者 PR 保存到目的地。

(2) 操作内容

```
STSMACH(long n) /* STS MACH,Rn */
{
    R[n]=MACH;
    PC+=2;
}
STSMACL(long n) /* STS MACL,Rn */
{
    R[n]=MACL;
    PC+=2;
}
STSPR(long n) /* STS PR,Rn */
{
    R[n]=PR;
    PC+=2;
}
STSMMACH(long n) /* STS.L MACH,@-Rn */
{
    R[n]-=4;
    Write_Long(R[n],MACH);
    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;
    }
```

(3) 使用例

```
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
```

6.4.60 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
SUB Rm,Rn	Rn-Rm→Rn	0011nnnnmmmm1000	1	—

(1) 说明

通用寄存器 Rn 的内容减去 Rm，结果保存到 Rn。使用 ADD #imm,Rn 进行和立即数的减法运算。

(2) 操作内容

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

(3) 使用例

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

6.4.61 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
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 和 T 位，结果保存到 Rn。根据运算结果将借位反映到 T 位。用于超过 32 位值的减法运算。

(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
```

6.4.62 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
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，结果保存到 Rn。当发生下溢时， 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
```

6.4.63 SWAP SWAP register halves：数据传送指令

高低位的交换

格式	操作概略	操作码	执行状态	T 位
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	—

(1) 说明

交换通用寄存器 Rm 内容的高低位，结果保存到 Rn。

当指定字节时，将 Rm 的 bit0 ～ bit7 和 bit8 ～ bit15 进行交换 （8 位）。将 Rm 的高 16 位传送到 Rn 的高 16 位。

当指定字时，将 Rm 的 bit0 ～ bit15 和 bit16 ～ bit31 进行交换 （16 位）。

(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
```

6.4.64 TAS Test And Set：逻辑运算指令

存储器的测试和位的置位

格式	操作概略	操作码	执行状态	T 位
TAS.B @Rn	(Rn) 为 0 时 1→T, 1→MSB of (Rn)	0100nnnn00011011	3	测试结果

(1) 说明

将通用寄存器 Rn 的内容作为地址，读该地址所指示的字节数据。当该数据是 0 时， T=1 ； 否则， T=0。然后，将 bit7 置 1 后写到相同的地址。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) 使用例

```
_LOOP   TAS.B       @R7       ;R7=1000
        BF          _LOOP     ; 循环到地址 1000 变为 0 为止。
```

6.4.65 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
TRAPA #imm	PC/SR→ 堆栈区 ,(imm×4+VBR)→PC	11000011iiiiiiii	5	—

(1) 说明

开始进行陷阱异常处理。即，将 PC 和 SR 保存到堆栈，转移到指定向量的内容所指示的地址。向量为 8 位立即数 （零扩展后）乘 4 后的存储器地址。 PC 为下一条指令的起始地址。

和 RTE 组合用于系统的调用。

(2) 注意

瑞萨科技的 “SuperH RISC engine 汇编程序”中，必须使用被放大的值 （×4）表述位移量的值。

(3) 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;
}
```

(4) 使用例

```
地址
VBR+H'80   .data.l   10000000      ;
           .......
            TRAPA      #H'20       ; 转移到地址 VBR+H'80 的内容所指示的地址。
             TST       #0,R0       ;← 从陷阱程序返回的地址
           .......                  （被压栈的 PC 内容）。
           .......
10000000   XOR        R0,R0        ;← 陷阱程序的入口
10000002   RTE                     ; 返回到上述 TST。
10000004   NOP                     ; 在 RTE 前执行。
```

6.4.66 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
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 的内容和 Rm 的逻辑与。当结果是 0 时，将 T 位置位；否则，清除 T 位。 Rn 的内容不变。

也可以是通用寄存器 R0 和 8 位立即数 （零扩展后）的逻辑与，或是 8 位存储器 （带变址的 GBR 间接寻址方式下）和 8 位立即数的逻辑与。 R0 或者存储器的内容不变。

(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
```

6.4.67 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
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 的内容和 Rm 的逻辑异或，结果保存到 Rn。

也可以是通用寄存器 R0 和 8 位立即数 （零扩展后）的逻辑异或，或是 8 位存储器 （带变址的 GBR 间接寻址方式下）和 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
```

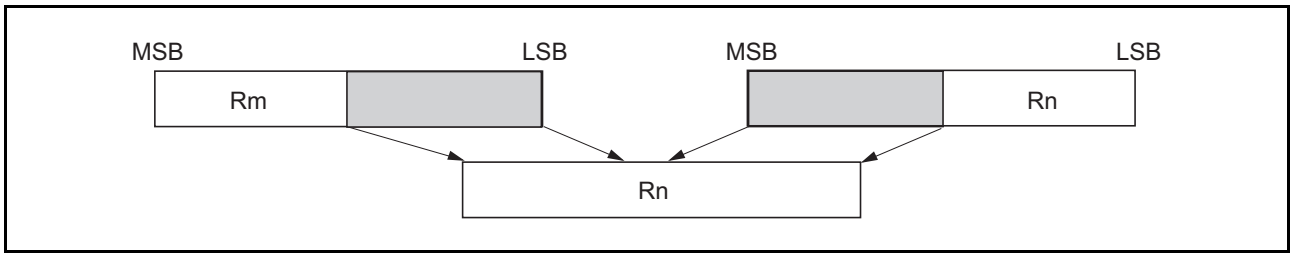
6.4.68 XTRCT eXTRaCT：数据传送指令

从连接寄存器的中间部分提取

格式	操作概略	操作码	执行状态	T 位
XTRCT Rm,Rn	Rm:Rn 的中间 32 位 →Rn	0010nnnnmmmm1101	1	—

(1) 说明

从连接通用寄存器 Rm 和 Rn 的 64 位内容中提取中间 32 位，结果保存到 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) 使用例

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

6.5 有关浮点指令和 FPU 的 CPU 指令

6.5.1 FABS Floating - point ABSolute value 浮点指令

浮点的绝对值

PR	格式	操作概略	操作码	执行状态	T 位
0	FABS FRn	|FRn|→FRn	1111nnnn01011101	1	—
1	FABS DRn	|DRn|→DRn	1111nnn001011101	1	—

(1) 说明

将浮点寄存器 FRn/DRn 的内容的最高位清 0，其结果保存到 FRn/DRn。
不更新 FPSCR 的 cause/flag 部分。

(2) 操作内容

```
void FABS (int n){
     FR[n] = FR[n] & 0x7fffffff;
     pc += 2;
}
/* 不依靠精度，进行相同的操作。 */
```

(3) 有可能发生的异常

无

6.5.2 FADD Floating - point ADD 浮点指令

浮点的加法

PR	格式	操作概略	操作码	执行状态	T 位
0	FADD FRm,FRn	FRn+FRm→FRn	1111nnnnmmmm0000	1	—
1	FADD DRm,DRn	DRn+DRm→DRn	1111nnn0mmm00000	6	—

(1) 说明

FPSCR.PR=0 时，对 FRn 和 FRm 内容的 2 个单精度浮点数进行加法运算，其结果保存到 FRn。

FPSCR.PR=1 时，对 DRn 和 DRm 内容的 2 个双精度浮点数进行加法运算，其结果保存到 DRn。

FPSCR.enable.O/U/I 被置位时，不论是否发生异常，都产生 FPU 异常陷阱。异常发生时， FPSCR.cause 和 FPSCR.flag 反映异常的正确信息， FRn/DRn 不被更新。请用软件进行恰当的处理。

(2) 操作内容

```
void FADD (int m,n)
{
      pc += 2;
      clear_cause();
      if((data_type_of(m) == sNaN) ||
         (data_type_of(n) == sNaN)) invalid(n);
      else if((data_type_of(m) == qNaN) ||
              (data_type_of(n) == qNaN)) qnan(n);
      else if((data_type_of(m) == DENORM) ||
              (data_type_of(n) == DENORM)) set_E();
      else switch (data_type_of(m)){
         case NORM: switch (data_type_of(n)){
             case NORM:      normal_faddsub(m,n,ADD); break;
             case PZERO:
             case NZERO:register_copy(m,n); break;
             default:        break;
      }            break;
      case PZERO: switch (data_type_of(n)){
      case NZERO:  zero(n,0); break;
      default:               break;
      }           break;
      case NZERO:  break;
      case PINF: switch (data_type_of(n)){
      case NINF:             invalid(n);     break;
          default:           inf(n,0);       break;
      }           break;
      case NINF: switch (data_type_of(n)){
      case PINF:             invalid(n);     break;
      default:               inf(n,1);       break;
      }           break;
```

```
            }
    }
```

FADD 特例

FRm,DRm	FRn,DRn						
	NORM	+0	–0	+INF	– INF	qNaN	sNaN
NORM	ADD				– INF		
+0		+0					
–0			–0				
+INF				+INF	Invalid		
– INF		– INF		Invalid	– INF		
qNaN						qNaN	
sNaN							Invalid

【注】 非规格化数的值作为零处理。

(3) 有可能发生的异常

无效运算 (Invalid operation)
上溢 (Overflow)
下溢 (Underflow)
不正确异常 (Inexact)

6.5.3 FCMP Floating - point CoMPare 浮点指令

浮点的比较

No.	PR	格式	操作概略	操作码	执行状态	T 位
1	0	FCMP/EQ FRm,FRn	(FRn==FRm)?1:0→T	1111nnnnmmmm0100	1	1/0
2	1	FCMP/EQ DRm,DRn	(DRn==DRm)?1:0→T	1111nnn0mmm00100	2	1/0
3	0	FCMP/GT FRm,FRn	(FRn>FRm)?1:0→T	1111nnnnmmmm0101	1	1/0
4	1	FCMP/GT DRm,DRn	(DRn>DRm)?1:0→T	1111nnn0mmm00101	2	1/0

(1) 说明

1. FPSCR.PR=0时，对FRn和FRm内容的2个单精度浮点数进行算术比较，相等时保存1否则保存0到T位。
2. FPSCR.PR=1时，对DRn和DRm内容的2个双精度浮点数进行算术比较，相等时保存1否则保存0到T位。
3. FPSCR.PR=0时，对FRn和FRm内容的2个单精度浮点数进行算术比较，FRn＞FRm时保存1否则保存0到T位。
4. FPSCR.PR=1时，对DRn和DRm内容的2个双精度浮点数进行算术比较，DRn＞DRm时保存1否则保存0到T位。

(2) 操作内容

```
void FCMP_EQ(int m,n) /* FCMP/EQ  FRm,FRn */
{
     pc += 2;
     clear_cause();
     if(fcmp_chk (m,n) == INVALID) fcmp_invalid();
     else if(fcmp_chk (m,n) == EQ)T = 1;
     else                          T = 0;
}
void FCMP_GT(int m,n) /* FCMP/GT  FRm,FRn */
{
     pc += 2;
     clear_cause();
     if ((fcmp_chk (m,n) == INVALID) ||
         (fcmp_chk (m,n) == UO)) fcmp_invalid();
     else if(fcmp_chk (m,n) == GT)  T = 1;
     else                          T = 0;
}
int fcmp_chk (int m,n)
{
     if((data_type_of(m) == sNaN) ||
        (data_type_of(n) == sNaN))        return(INVALID);
     else if((data_type_of(m) == qNaN) ||
             (data_type_of(n) == qNaN))      return(UO);
     else switch(data_type_of(m)){
```

```
            case NORM:     switch(data_type_of(n)){
                  case PINF   :return(GT);break;
                  case NINF   :return(LT);break;
                  default:                break;
                  }   break;
            case PZERO:
            case NZERO:    switch(data_type_of(n)){
                  case PZERO  :
                  case NZERO  :return(EQ);break;
                  }      break;
            case PINF :    switch(data_type_of(n)){
                  case PINF   :return(EQ);break;
                  default:return(LT);break;
                  }      break;
            case NINF :    switch(data_type_of(n)){
                  case NINF   :return(EQ);break;
                  default:return(GT);break;
                  }      break;
      }
      if(FPSCR_PR == 0) {
           if(FR[n] == FR[m])        return(EQ);
           else if(FR[n] > FR[m])    return(GT);
           else                      return(LT);
      }else {
           if(DR[n>>1] == DR[m>>1])     return(EQ);
           else if(DR[n>>1] > DR[m>>1]) return(GT);
           else                         return(LT);
      }
}
void fcmp_invalid()
{
      set_V(); T = 0; if((FPSCR & ENABLE_V) == 1) fpu_exception_trap();
}
```

FCMP 特例

FCMP/EQ	FRn,DRn						
FRm,DRm	NORM	+0	–0	+INF	– INF	qNaN	sNaN
NORM	CMP						
+0		EQ					
–0							
+INF				EQ			
– INF					EQ		
qNaN						!EQ	
sNaN							Invalid

【注】 非规格化数的值作为零处理。

FCMP/GT	FRn,DRn						
FRm,DRm	NORM	+0	-0	+INF	-INF	qNaN	sNaN
NORM	CMP			GT	!GT		
+0		!GT					
-0							
+INF	!GT			!GT			
-INF	GT				!GT		
qNaN						UO	
sNaN							Invalid

【注】 非规格化数的值作为零处理。
UO是无序的。无序时作为!GT处理。

(3) 有可能发生的异常

无效运算 (Invalid operation)

6.5.4 FCNVDS Floating - point CoNVert Double to Single precision 浮点指令

将双精度转换为单精度

PR	格式	操作概略	操作码	执行状态	T 位
0	—	—	—	—	—
1	FCNVDS DRm,FPUL	(float)DRm →FPUL	1111mmm010111101	2	—

(1) 说明

FPSCR.PR=1 时，将 DRm 内的双精度浮点数转换为单精度浮点数并保存到 FPUL。

FPSCR.enable.O/U/I 被置位时，不论是否发生异常，都产生 FPU 异常陷阱。异常发生时， FPSCR.cause 和 FPSCR.flag 反映异常的正确信息， FPUL 不被更新。请用软件进行恰当的处理。

FPSCR.PR=0 时，为非法指令。

(2) 操作内容

```
void FCNVDS(int m,float *FPUL){
     case((FPSCR.PR){
          0:  undefined_operation();   /* reserved */
          1:  fcnvds(m, *FPUL);  break;  /* FCNVDS */
     }
}
void fcnvds(int m, float *FPUL)
{
     pc += 2;
     clear_cause();
     case(data_type_of(m)){
          NORM  :
          PZERO :
          NZERO :            normal_ fcnvds(m, *FPUL);  break;
          PINF  :            *FPUL = 0x7f800000; break;
          NINF  :            *FPUL = 0xff800000; break;
          qNaN  :            *FPUL = 0x7fbfffff; break;
          sNaN  :            set_V();
                             if((FPSCR & ENABLE_V) == 0) *FPUL = 0x7fbfffff;
                             else fpu_exception_trap();   break;
     }
}
void normal_fcnvds(int m,float *FPUL)
{
int sign;
float abs;
union {
      float f;
      int l;
}     dstf,tmpf;
```

```
        union {
              double d;
              int l[2];
        }     dstd;
              dstd.d = DR[m>>1];
              if(dstd.l[1] & 0x1fffffff)) set_I();
              if(FPSCR_RM == 1) dstd.l[1] &= 0xe0000000; /* round toward zero */
              dstf.f = dstd.d;
              check_single_exception(FPUL, dstf.f);
  }
```

FCNVDS 特例

FRn	+NORM	– NORM	+0	–0	+INF	– INF	qNaN	sNaN
FCNVDS(FRn FPUL)	FCNVDS	FCNVDS	+0	–0	+INF	– INF	qNaN	Invalid

【注】 非规格化数的值作为零处理。

(3) 有可能发生的异常

无效运算 (Invalid operation)
上溢 (Overflow)
下溢 (Underflow)
不正确异常 (Inexact)

6.5.5 FCNVSD Floating - point CoNVert Single to Double precision 浮点指令

将单精度转换为双精度

PR	格式	操作概略	操作码	执行状态	T 位
0	—	—	—	—	—
1	FCNVSD FPUL, DRn	(double)FPUL→DRn	1111nnn010101101	2	—

(1) 说明

FPSCR.PR=1 时，将 FPUL 的内容解释为单精度浮点数，并转换为双精度浮点数，其结果保存到 DRn。
FPSCR.PR=0 时，为非法指令。

(2) 操作内容

```
void FCNVSD(int n, float *FPUL){
     pc += 2;
     clear_cause();
     case((FPSCR_PR){
          0:  undefined_operation();    /* reserved */
          1:  fcnvsd (n, *FPUL);  break;  /* FCNVSD */
     }
}
void fcnvsd(int n, float *FPUL)
{
     case(fpul_type(*FPUL)){
          PZERO :
          NZERO :
          PINF  :
          NINF  :DR[n>>1] = *FPUL; break;
          qNaN  :qnan(n);          break;
          sNaN  :invalid(n);       break;
     }
}
int fpul_type(int *FPUL)
{
int abs;
          abs = *FPUL & 0x7fffffff;
      if(abs < 0x00800000){
        if((FPSCR_DN == 1) || (abs == 0x00000000)){
          if(sign_of(src) == 0)   return(PZERO);
          else                    return(NZERO);
        }
        else                      return(DENORM);
      }
      else if(abs < 0x7f800000)   return(NORM);
      else if(abs == 0x7f800000) {
```

```
            if(sign_of(src) == 0)   return(PINF);
            else                    return(NINF);
        }
        else if(abs < 0x7fc00000)   return(qNaN);
        else                        return(sNaN);
    }
```

FCNVSD 特例

FRn	+NORM	– NORM	+0	–0	+INF	– INF	qNaN	sNaN
FCNVSD(FPUL FRn)	+NORM	– NORM	+0	–0	+INF	– INF	qNaN	Invalid

【注】 非规格化数的值作为零处理。

(3) 有可能发生的异常

无效运算 (Invalid operation)

6.5.6 FDIV Floating - point DIVide 浮点指令

浮点的除法

PR	格式	操作概略	操作码	执行状态	T 位
0	FDIV FRm,FRn	FRn/FRm→FRn	1111nnnnmmmm0011	10	—
1	FDIV DRm,DRn	DRn/DRm→DRn	1111nnn0mmm00011	23	—

(1) 说明

FPSCR.PR=0 时，对 FRn 和 FRm 内容的 2 个单精度浮点数进行除法运算，其结果保存到 FRn。

FPSCR.PR=1 时，对 DRn 和 DRm 内容的 2 个双精度浮点数进行除法运算，其结果保存到 DRn。

FPSCR.enable.O/U/I 被置位时，不论是否发生异常，都产生 FPU 异常陷阱。异常产生时， FPSCR.cause 和 FPSCR.flag 反映异常的正确信息， FRn/DRn 不被更新。请用软件进行恰当的处理。

(2) 操作内容

```
void FDIV(int m,n)     /* FDIV FRm,FRn */
{
     pc += 2;
     clear_cause();
     if((data_type_of(m) == sNaN) ||
        (data_type_of(n) == sNaN)) invalid(n);
     else if((data_type_of(m) == qNaN) ||
             (data_type_of(n) == qNaN)) qnan(n);
     else switch (data_type_of(m)){
              case NORM: switch (data_type_of(n)){
              case PINF:
              case NINF:              inf(n,sign_of(m)^sign_of(n));break;
              case PZERO:
              case NZERO:zero(n,sign_of(m)^sign_of(n));break;
              case DENORM:            set_E();         break;
              default:                normal_fdiv(m,n);break;
 }      break;
      case PZERO: switch (data_type_of(n)){
           case PZERO:
           case NZERO:invalid(n);break;
           case PINF:
           case NINF:             break;
           default:               dz(n,sign_of(m)^sign_of(n));break;
                    }             break;
      case NZERO: switch (data_type_of(n)){
           case PZERO:
           case NZERO:invalid(n);  break;
           case PINF:              inf(n,1);break;
           case NINF:             inf(n,0);break;
           default:               dz(FR[n],sign_of(m)^sign_of(n)); break;
```

```
                }      break;
            case PINF :
            case NINF : switch (data_type_of(n)){
                case PINF:
                case NINF: invalid(n);     break;
                default:   zero(n,sign_of(m)^sign_of(n));break
                }      break;
        }
}
void normal_fdiv(int m,n)
{
union {
        float f;
        int l;
}     dstf,tmpf;
union {
        double d;
        int l[2];
}     dstd,tmpd;
union {
        int double x;
        int l[4];
}     tmpx;
        if(FPSCR_PR == 0) {
        tmpf.f = FR[n]; /* save destination value */
        dstf.f /= FR[m]; /* round toward nearest or even */
        tmpd.d = dstf.f; /* convert single to double */
        tmpd.d *= FR[m];
        if(tmpf.f != tmpd.d) set_I();
        if((tmpf.f < tmpd.d) && (SPSCR_RM == 1))
        dstf.l -= 1; /* round toward zero */
        check_single_exception(&FR[n], dstf.f);
        } else {
                tmpd.d = DR[n>>1]; /* save destination value */
                dstd.d /= DR[m>>1]; /* round toward nearest or even */
                tmpx.x = dstd.d; /* convert double to int double */
                tmpx.x *= DR[m>>1];
        if(tmpd.d != tmpx.x) set_I();
        if((tmpd.d < tmpx.x) && (SPSCR_RM == 1)) {
                dstd.l[1] -= 1; /* round toward zero */
        if(dstd.l[1] == 0xffffffff) dstd.l[0] -= 1;
        }
        check_double_exception(&DR[n>>1], dstd.d);
        }
}
```

FDIV 特例

FRm,DRm	FRn,DRn						
	NORM	+0	–0	+INF	– INF	qNaN	sNaN
NORM	DIV	0		INF			
+0	DZ	Invalid		+INF	– INF		
–0				– INF	+INF		
+INF	0	+0	–0	Invalid			
– INF		–0	+0				
qNaN						qNaN	
sNaN							Invalid

【注】 非规格化数的值作为零处理。

(3) 有可能发生的异常

无效运算 (Invalid operation)
被零除 (Divide by zero)
上溢 (Overflow)
下溢 (Underflow)
不正确异常 (Inexact)

6.5.7 FLDI0 Floating - point LoaD Immediate 0.0 浮点指令

0.0 加载

PR	格式	操作概略	操作码	执行状态	T 位
0	FLDI0 FRn	0x00000000→FRn	1111nnnn10001101	1	—
1	—	—	—	—	—

(1) 说明

FPSCR.PR=0 时，将浮点数的 0.0(0x00000000) 保存到 FRn。
FPSCR.PR=1 时，为非法指令。

(2) 操作内容

```
void FLDI0(int n)
{
     FR[n] = 0x00000000;
     pc += 2;
}
```

(3) 有可能发生的异常

无

6.5.8 FLDI1 Floating - point LoaD Immediate 1.0 浮点指令

1.0 加载

PR	格式	操作概略	操作码	执行状态	T 位
0	FLDI1 FRn	0x3F800000→FRn	1111nnnn10011101	1	—
1	—	—	—	—	—

(1) 说明

FPSCR.PR=0 时，将浮点数的 1.0(0x3F800000) 保存到 FRn。
FPSCR.PR=1 时，为非法指令。

(2) 操作内容

```
void FLDI1(int n)
{
     FR[n] = 0x3F800000;
     pc += 2;
}
```

(3) 有可能发生的异常

无

6.5.9 FLDS Floating - point LoaD to System register 浮点指令

向系统寄存器的传送

格式	操作概略	操作码	执行状态	T 位
FLDS FRm,FPUL	FRm → FPUL	1111mmmm00011101	1	—

(1) 说明

将浮点寄存器 FRm 的内容保存到系统寄存器 FPUL。

(2) 操作内容

```
void FLDS(int m,float *FPUL)
{
     *FPUL = FR[m];
     pc += 2;
}
```

(3) 有可能发生的异常

无

6.5.10 FLOAT Floating - point convert from integer 浮点指令

将整数转换为浮点数

PR	格式	操作概略	操作码	执行状态	T 位
0	FLOAT FPUL,FRn	(float)FPUL→FRn	1111nnnn0010110	1	—
1	FLOAT FPUL,DRn	(double)FPUL→DRn	11111nnn000101101	2	—

(1) 说明

FPSCR.PR=0 时，将 FPUL 的内容视为 32 位整数，并转换为单精度浮点数，其结果保存到 FRn。

FPSCR.PR=1 时，将 FPUL 的内容视为 32 位整数，并转换为双精度浮点数，其结果保存到 DRn。

FPSCR.enable.I=1 和 FPCSR.PR=0 时，不论是否发生异常，都产生 FPU 异常陷阱。异常产生时，FPSCR.cause 和 FPSCR.flag 反映异常的正确信息， FRn/DRn 不被更新。请用软件进行恰当的处理。

(2) 操作内容

```
        void FLOAT(int n,float *FPUL)
        {
        union {
              double d;
              int l[2];
        }     tmp;
              pc += 2;
              clear_cause();
              if(FPSCR.PR==0){
                  FR[n] = *FPUL; /* convert from integer to float */
                  tmp.d = *FPUL;
                  if(tmp.l[1] & 0x1fffffff) inexact();
              } else {
                  DR[n>>1] = *FPUL; /* convert from integer to double */
              }
  }
```

(3) 有可能发生的异常

不正确异常 (Inexact)：FPSCR.PR =1 时不发生。

6.5.11 FMAC Floating - point Multiply and Accumulate 浮点指令

浮点的乘法累加

PR	格式	操作概略	操作码	执行状态	T 位
0	FMAC FR0,FRm,FRn	FR0*FRm+FRn→FRn	1111nnnnmmmm1110	1	—
1	—	—	—	—	—

(1) 说明

FPSCR.PR=0 时，对 FR0 和 FRm 内容的 2 个单精度浮点数进行乘法运算，并且对 FRn 的内容进行加法运算，其结果保存到 FRn。

FPSCR.enable.O/U/I 被置位时，不论是否发生异常，都产生 FPU 异常陷阱。异常发生时， FPSCR.cause 和 FPSCR.flag 反映异常的正确信息， FRn 不被更新。请用软件进行恰当的处理。

FPSCR.PR=1 时，为非法指令。

(2) 操作内容

```
void FMAC(int m,n)
{
      pc += 2;
      clear_cause();
      if(FPSCR_PR == 1) undefined_operation();
      else if((data_type_of(0) == sNaN) ||
               (data_type_of(m) == sNaN) ||
               (data_type_of(n) == sNaN)) invalid(n);
      else if((data_type_of(0) == qNaN) ||
              (data_type_of(m) == qNaN)) qnan(n);
      else if((data_type_of(0) == DENORM) ||
              (data_type_of(m) == DENORM)) set_E();
      else switch (data_type_of(0){
              case NORM: switch (data_type_of(m)){
              case PZERO:
              case NZERO: switch (data_type_of(n)){
              case qNaN:          qnan(n);          break;
              case PZERO:
              case NZERO:zero(n,sign_of(0)^ sign_of(m)^sign_of(n));  break;
              default:          break;
              }
                 case PINF:
      case NINF: switch (data_type_of(n)){
              case qNaN: qnan(n);       break;
              case PINF:
              case NINF: if(sign_of(0)^ sign_of(m)^sign_of(n))invalid(n);
              else inf(n,sign_of(0)^ sign_of(m));break;
              default:   inf(n,sign_of(0)^ sign_of(m));        break;
              }
```

```
            case NORM: switch (data_type_of(n)){
                    case qNaN: qnan(n);      break;
                    case PINF:
                    case NINF: inf(n,sign_of(n));break;
                    case PZERO:
                    case NZERO:
                    case NORM: normal_fmac(m,n);break;
      }        break;
      case PZERO:
      case NZERO: switch (data_type_of(m)){
                    case PINF:
                    case NINF:               invalid(n);break;
                    case PZERO:
                    case NZERO:
                    case NORM: switch (data_type_of(n)){
                    case qNaN:               qnan(n);break;
                    case PZERO:
                    case NZERO: zero(n,sign_of(0)^ sign_of(m)^sign_of(n));  break;
                    default:                 break;
      }        break;
}            break;
      case PINF :
      case NINF : switch (data_type_of(m)){
                    case PZERO:
                    case NZERO: invalid(n);   break;
                    default: switch (data_type_of(n)){
      case qNaN: qnan(n);                     break;
      default:     inf(n,sign_of(0)^sign_of(m)^sign_of(n));break
      }    break;
      }      break;
   }
            }
}
void normal_fmac(int m,n)
{
union {
      int double x;
      int l[4];
      }     dstx,tmpx;
float dstf,srcf;
      if((data_type_of(n) == PZERO)|| (data_type_of(n) == NZERO))
      srcf = 0.0; /* flush denormalized value */
      else  srcf = FR[n];
      tmpx.x = FR[0]; /* convert single to int double */
      tmpx.x *= FR[m]; /* exact product */
      dstx.x = tmpx.x + srcf;
```

```
        if(((dstx.x == srcf) && (tmpx.x != 0.0)) ||
              ((dstx.x == tmpx.x) && (srcf != 0.0))) {
              set_I();
              if(sign_of(0)^ sign_of(m)^ sign_of(n))  {
              dstx.l[3] -= 1; /* correct result */
        if(dstx.l[3] == 0xffffffff) dstx.l[2] -= 1;
              if(dstx.l[2] == 0xffffffff) dstx.l[1] -= 1;
              if(dstx.l[1] == 0xffffffff) dstx.l[0] -= 1;
              }
              else          dstx.l[3] |= 1;
        }
        if((dstx.l[1] & 0x01ffffff) || dstx.l[2] || dstx.l[3]) set_I();
              if(FPSCR_RM == 1) {
              dstx.l[1] &= 0xfe000000; /* round toward zero */
              dstx.l[2]  = 0x00000000;
              dstx.l[3]  = 0x00000000;
        }
              dstf = dstx.x;
        check_single_exception(&FR[n],dstf);
  }
```

FMAC 特例

FRn	FR0	FRm							
		+Norm	– Norm	+0	–0	+INF	– INF	qNaN	sNaN
Norm	Norm	MAC				INF			
	0					Invalid			
	INF	INF		Invalid		INF			
+0	Norm	MAC							
	0				+0	Invalid			
	INF	INF		Invalid		INF			
–0	+Norm	MAC		+0	–0	+INF	– INF		
	– Norm			–0	+0	– INF	+INF		
	+0	+0	–0	+0	–0	Invalid			
	–0	–0	+0	–0	+0				
	INF	INF		Invalid		INF			
+INF	+Norm	+INF					Invalid		
	– Norm						+INF		
	0					Invalid			
	+INF			Invalid		+INF			
	– INF	Invalid	+INF				+INF		
– INF	+Norm	– INF					– INF		
	– Norm								
	0								
	+INF	Invalid		Invalid			-INF		
	– INF	– INF				– INF	Invalid		
qNaN	0					Invalid			
	INF			Invalid					
	Norm								
!sNaN	qNaN							qNaN	
All types	sNaN								
SNaN	all types								Invalid

【注】 非规格化数的值作为零处理。

(3) 有可能发生的异常

无效运算 (Invalid operation)
上溢 (Overflow)
下溢 (Underflow)
不正确异常 (Inexact)

6.5.12 FMOV Floating - point MOVe 浮点指令

浮点的传送

No.	SZ	格式	操作概略	操作码	执行状态	T 位
1	0	FMOV FRm,FRn	FRm→FRn	1111nnnnmmmm1100	1	—
2	1	FMOV DRm,DRn	DRm→DRn	1111nnn0mmm01100	2	—
3	0	FMOV.S FRm,@Rn	FRm→(Rn)	1111nnnnmmmm1010	1	—
4	1	FMOV.D DRm,@Rn	DRm→(Rn)	1111nnnnmmm01010	2	—
5	0	FMOV.S @Rm,FRn	(Rm)→FRn	1111nnnnmmmm1000	1	—
6	1	FMOV.D @Rm,DRn	(Rm)→DRn	1111nnn0mmmm1000	2	—
7	0	FMOV.S @Rm+,FRn	(Rm)→FRn,Rm+=4	1111nnnnmmmm1001	1	—
8	1	FMOV.D @Rm+,DRn	(Rm)→DRn,Rm+=8	1111nnn0mmmm1001	2	—
9	0	FMOV.S FRm,@-Rn	Rn-=4,FRm→(Rn)	1111nnnnmmmm1011	1	—
10	1	FMOV.D DRm,@-Rn	Rn-=8,DRm→(Rn)	1111nnnnmmm01011	2	—
11	0	FMOV.S @(R0,Rm),FRn	(R0+Rm)→FRn	1111nnnnmmmm0110	1	—
12	1	FMOV.D @(R0,Rm),DRn	(R0+Rm)→DRn	1111nnn0mmmm0110	2	—
13	0	FMOV.S FRm, @(R0,Rn)	FRm→(R0+Rn)	1111nnnnmmmm0111	1	—
14	1	FMOV.D DRm, @(R0,Rn)	DRm→(R0+Rn)	1111nnnnmmm00111	2	—

(1) 说明

1. 将 FRm 的内容传送到 FRn。
2. 将 DRm 的内容传送到 DRn。
3. 将 FRm 的内容传送到 Rn 所指向地址的存储器。
4. 将 DRm 的内容传送到 Rn 所指向地址的存储器。
5. 将 Rm 所指向地址的存储器的内容传送到 FRn。
6. 将 Rm 所指向地址的存储器的内容传送到 DRn。
7. 将 Rm 所指向地址的存储器的内容传送到 FRn，并且给 Rm 加 4。
8. 将 Rm 所指向地址的存储器的内容传送到 DRn，并且给 Rm 加 8。
9. Rn 减 4，并且将 FRm 的内容传送到其 Rn 所指向地址的存储器。
10. Rn 减 8，并且将 DRm 的内容传送到其 Rn 所指向地址的存储器。
11. 将 (R0+Rm) 所指向地址的存储器的内容传送到 FRn。
12. 将 (R0+Rm) 所指向地址的存储器的内容传送到 DRn。
13. 将 FRm 的内容传送到 (R0+Rn) 所指向地址的存储器。
14. 将 DRm 的内容传送到 (R0+Rn) 所指向地址的存储器。

(2) 操作内容

```
void FMOV(int m,n)                /* FMOV FRm,FRn */
{
     FR[n] = FR[m];
     pc += 2;
}
void FMOV_DR(int m,n)            /* FMOV DRm,DRn */
{
     DR[n>>1] = DR[m>>1];
     pc += 2;
}
void FMOV_STORE(int m,n)        /* FMOV.S FRm,@Rn */
{
     store_int(FR[m],R[n]);
     pc += 2;
}
void FMOV_STORE_DR(int m,n)    /* FMOV.D DRm,@Rn */
{
     store_quad(DR[m>>1],R[n]);
     pc += 2;
}
 void FMOV_LOAD(int m,n)        /* FMOV.S @Rm,FRn */
{
     load_int(R[m],FR[n]);
     pc += 2;
}
void FMOV_LOAD_DR(int m,n)     /* FMOV.D @Rm,DRn */
{
     load_quad(R[m],DR[n>>1]);
     pc += 2;
}
void FMOV_RESTORE(int m,n)      /* FMOV.S @Rm+,FRn */
{
     load_int(R[m],FR[n]);
     R[m] += 4;
     pc += 2;
}
void FMOV_RESTORE_DR(int m,n) /* FMOV.D @Rm+,DRn */
{
     load_quad(R[m],DR[n>>1]) ;
     R[m] += 8;
     pc += 2;
}
void FMOV_SAVE(int m,n)         /* FMOV.S FRm,@-Rn */
{
```

```
        store_int(FR[m],R[n]-4);
        R[n] -= 4;
        pc += 2;
    }
    void FMOV_SAVE_DR(int m,n)    /* FMOV.D DRm,@-Rn */
    {
        store_quad(DR[m>>1],R[n]-8);
        R[n] -= 8;
        pc += 2;
    }
    void FMOV_INDEX_LOAD(int m,n)  /* FMOV.S @(R0,Rm),FRn */
    {
        load_int(R[0] + R[m],FR[n]);
        pc += 2;
    }
    void FMOV_INDEX_LOAD_DR(int m,n) /*FMOV.D @(R0,Rm),DRn */
    {
        load_quad(R[0] + R[m],DR[n>>1]);
        pc += 2;
    }
    void FMOV_INDEX_STORE(int m,n)  /*FMOV.S FRm,@(R0,Rn)*/
    {
        store_int(FR[m], R[0] + R[n]);
        pc += 2;
    }
    void FMOV_INDEX_STORE_DR(int m,n)/*FMOV.D DRm,@(R0,Rn)*/
    {
        store_quad(DR[m>>1], R[0] + R[n]);
        pc += 2;
    }
```

(3) 有可能发生的异常

地址错误

6.5.13 FMUL Floating - point MULtiply 浮点指令

浮点乘法

PR	格式	操作概略	操作码	执行状态	T 位
0	FMUL FRm,FRn	FRn*FRm→FRn	1111nnnnmmmm0010	1	—
1	FMUL DRm,DRn	DRn*DRm→DRn	1111nnn0mmm00010	6	—

(1) 说明

FPSCR.PR=0 时，对 FRn 和 FRm 内容的 2 个单精度浮点数进行乘法运算，其结果保存到 FRn。

FPSCR.PR=1 时，对 DRn 和 DRm 内容的 2 个双精度浮点数进行乘法运算，其结果保存到 DRn。

FPSCR.enable.O/U/I 被置位时，不论是否发生异常，都产生 FPU 异常陷阱。异常发生时， FPSCR.cause 和 FPSCR.flag 反映异常的正确信息， FRn/DRn 不被更新。请用软件进行恰当的处理。

(2) 操作内容

```
        void FMUL(int m,n)
        {
            pc += 2;
              clear_cause();
              if((data_type_of(m) == sNaN) ||
                 (data_type_of(n) == sNaN)) invalid(n);
              else if((data_type_of(m) == qNaN) ||
                      (data_type_of(n) == qNaN)) qnan(n);
              else switch (data_type_of(m){
                case NORM: switch (data_type_of(n)){
                case PZERO:
                case NZERO:zero(n,sign_of(m)^sign_of(n));  break;
                case PINF:
                case NINF    inf(n,sign_of(m)^sign_of(n));  break;
                default:     normal_fmul(m,n);break;
        }       break;
              case PZERO:
              case NZERO: switch (data_type_of(n)){
                case PINF:
                case NINF:          invalid(n);break;
                default:             zero(n,sign_of(m)^sign_of(n));break;
            }      break;
              case PINF :
              case NINF : switch (data_type_of(n)){
                case PZERO:
                case NZERO:invalid(n);break;
                   default:
            inf(n,sign_of(m)^sign_of(n));break
            }      break;
            }
  }
```

FMUL 特例

FRm,DRm	FRn,DRn						
	NORM	+0	–0	+INF	– INF	qNaN	sNaN
NORM	MUL	0		INF			
+0	0	+0	-0	Invalid			
–0		–0	+0				
+INF	INF	Invalid		+INF	– INF		
– INF				– INF	+INF		
qNaN						qNaN	
sNaN							Invalid

【注】 非规格化数的值作为零处理。

(3) 有可能发生的异常

无效运算 (Invalid operation)
上溢 (Overflow)
下溢 (Underflow)
不正确异常 (Inexact)

6.5.14 FNEG Floating - point NEGate value 浮点指令

浮点的符号取反

PR	格式	操作概略	操作码	执行状态	T 位
0	FNEG FRn	– FRn→FRn	1111nnnn01001101	1	—
1	FNEG DRn	– DRn→DRn	1111nnn001001101	1	—

(1) 说明

对浮点寄存器 FRn/DRn 内容的最高位 （符号位）进行取反，其结果保存到 FRn/DRn。
不更新 FPSCR 的 cause/flag 部分。

(2) 操作内容

```
void FNEG (int n){
     FR[n] = -FR[n];
     pc += 2;
}

/* 不依靠精度，进行相同的操作。 */
```

(3) 有可能发生的意外

无

6.5.15 FSCHG Sz-bit CHanGe 浮点指令

SZ 位取反

PR	格式	操作概略	操作码	执行状态	T 位
0	FSCHG	FPSCR.SZ=~FPSCR.SZ	1111001111111101	1	—
1	—	—	—	—	—

(1) 说明

FPSCR.PR=0 时，对浮点状态寄存器 FPSCR 的 SZ 位进行取反。如果改变 FPSCR 的 SZ 位， FMOV 指令的数据传送转换为 1 个或者 1 对单精度数据。 FPSCR.SZ=0 时， FMOV 指令传送 1 个单精度数据。 FPSCR.SZ=1 时， FMOV 指令按对传送 2 个单精度数据。

FPSCR.PR=1 时，为非法指令。

(2) 操作内容

```
void FSCHG()   /* FSCHG */
{
      if(FPSCR_PR == 0){
              FPSCR ^= 0x00100000; /* bit 20 */
              PC += 2;
      }
      else undefined_operation();
}
```

(3) 有可能发生的异常

无

6.5.16 FSQRT Floating - point SQuare RooT 浮点指令

浮点平方根

PR	格式	操作概略	操作码	执行状态	T 位
0	FSQRT FRn	$\sqrt{FRn} \rightarrow FRn$	1111nnnn01101101	9	—
1	FSQRT DRn	$\sqrt{DRn} \rightarrow DRn$	1111nnnn01101101	22	—

(1) 说明

FPSCR.PR=0 时，求 FRn 内容的单精度浮点数的算术平方根，其结果保存到 FRn。

FPSCR.PR=1 时，求 DRn 内容的双精度浮点数的算术平方根，其结果保存到 DRn。

FPSCR.enable.I 被置位时，不论是否发生异常，都产生 FPU 异常陷阱。异常发生时， FPSCR.cause 和 FPSCR.flag 反映异常的正确信息， FRn/DRn 不被更新。请用软件进行恰当的处理。

(2) 操作内容

```
        void FSQRT(int n){
              pc += 2;
              clear_cause();
              switch(data_type_of(n)){
              case NORM  : if(sign_of(n) == 0)normal_ fsqrt(n);
                           else  invalid(n);break;
              case PZERO :
              case NZERO :
              case PINF  :     break;
              case NINF  :     invalid(n);break;
              case qNaN  :     qnan(n);   break;
              case sNaN  :     invalid(n);break;
        }
  }
  void normal_fsqrt(int n)
  {
  union {
        float f;
        int l;
  }     dstf,tmpf;
  union {
        double d;
        int l[2];
   }    dstd,tmpd;
   union {
        int double x;
        int l[4];
  }           tmpx;
              if(FPSCR_PR == 0) {
                 tmpf.f = FR[n]; /* save destination value */
```

```
                dstf.f = sqrt(FR[n]); /* round toward nearest or even */
                tmpd.d = dstf.f; /* convert single to double */
                tmpd.d *= dstf.f;
                if(tmpf.f != tmpd.d) set_I();
                if((tmpf.f < tmpd.d) && (SPSCR_RM == 1))
                   dstf.l -= 1; /* round toward zero */
                if(FPSCR & ENABLE_I)fpu_exception_trap();
                else       FR[n] = dstf.f;
        } else {
                tmpd.d = DR[n>>1]; /* save destination value */
                dstd.d = sqrt(DR[n>>1]); /* round toward nearest or even */
                tmpx.x = dstd.d; /* convert double to int double */
                tmpx.x *= dstd.d;
                if(tmpd.d != tmpx.x) set_I();
                if((tmpd.d < tmpx.x) && (SPSCR_RM == 1)) {
                   dstd.l[1] -= 1; /* round toward zero */
                   if(dstd.l[1] == 0xffffffff) dstd.l[0] -= 1;
             }
                if(FPSCR & ENABLE_I)fpu_exception_trap();
                else      DR[n>>1] = dstd.d;
        }
}
```

FSQRT 特例

FRn	+NORM	– NORM	+0	–0	+INF	– INF	qNaN	sNaN
FSQRT(FRn)	SQRT	Invalid	+0	–0	+INF	Invalid	qNaN	Invalid

【注】 非规格化数的值作为零处理。

(3) 有可能发生的异常

无效运算 (Invalid operation)
不正确异常 (Inexact)

6.5.17 FSTS Floating - point Store System register 浮点指令

从系统寄存器的传送

格式	操作概略	操作码	执行状态	T 位
FSTS FPUL,FRn	FPUL→FRn	1111nnnn00001101	1	—

(1) 说明

将系统寄存器 FPUL 的内容传送到浮点寄存器 FRn。

(2) 操作内容

```
void FSTS(int n, float *FPUL)
{
     FR[n] = *FPUL;
     pc += 2;
}
```

(3) 可能发生的异常

无

6.5.18 FSUB Floating - point SUBtract 浮点指令

浮点的减法

PR	格式	操作概略	操作码	执行状态	T 位
0	FSUB FRm,FRn	FRn-FRm→FRn	1111nnnnmmmm0001	1	—
1	FSUB DRm,DRn	DRn-DRm→DRn	1111nnn0mmm00001	6	—

(1) 说明

FPSCR.PR=0 时，对 FRn 和 FRm 内容的 2 个单精度浮点数进行减法运算，其结果保存到 FRn。

FPSCR.PR=1 时，对 DRn 和 DRm 内容的 2 个双精度浮点数进行减法运算，其结果保存到 DRn。

FPSCR.enable.O/U/I 被置位时，不论是否发生异常，都产生 FPU 异常陷阱。异常发生时， FPSCR.cause 和 FPSCR.flag 反映异常的正确信息， FRn/DRn 不被更新。请用软件进行恰当的处理。

(2) 操作内容

```
        void FSUB (int m,n)
        {
              pc += 2;
              clear_cause();
              if((data_type_of(m) == sNaN) ||
              (data_type_of(n) == sNaN)) invalid(n);
              else if((data_type_of(m) == qNaN) ||
                      (data_type_of(n) == qNaN)) qnan(n);
              else switch (data_type_of(m)){
                      case NORM: switch  pe_of(n)){
                           case NORM:normal_faddsub(m,n,SUB); break;
                           case PZERO:
                           case NZERO:register_copy(m,n); FR[n] = -FR[n];break;
                           default:  break;
              }            break;
              case PZERO:           break;
              case NZERO: switch
                 case NZERO:zero(n,0); break;
                 default:           break;
              }            break;
              case PINF: switch (data_type_of(n)){
                 case PINF:         invalid(n);     break;
                 default:           inf(n,1);       break;
              }            break;
              case NINF: switch (data_type_of(n)){
                 case NINF:         invalid(n);     break;
                 default:           inf(n,0);       break;
              }            break;
        }
  }
```

FSUB 特例

FRm,DRm	FRn,DRn						
	NORM	+0	–0	+INF	– INF	qNaN	sNaN
NORM	SUB			+INF	– INF		
+0			–0				
–0		+0					
+INF	– INF			Invalid			
– INF	+INF				Invalid		
qNaN						qNaN	
sNaN							Invalid

【注】 非规格化数的值作为零处理。

(3) 有可能发生的异常

无效运算 (Invalid operation)
上溢 (Overflow)
下溢 (Underflow)
不正确异常 (Inexact)

6.5.19 FTRC Floating - point Truncate and Convert to integer 浮点指令转换为整数

PR	格式	操作概略	操作码	执行状态	T 位
0	FTRC FRm,FPUL	(long)FRm→FPUL	1111mmmm00111101	1	—
1	FTRC DRm,FPUL	(long)DRm→FPUL	1111mmm000111101	2	—

(1) 说明

FPSCR.PR=0 时，将 FRm 内容的单精度浮点数转换为 32 位整数，其结果保存到 FPUL。
FPSCR.PR=1 时，将 DRm 内容的双精度浮点数转换为 32 位整数，其结果保存到 FPUL。
舍入模式总是舍去。

(2) 操作内容

```
#define N_INT_SINGLE_RANGE 0xcf000000 & 0x7fffffff  /* -1.000000 * 2^31 */
#define P_INT_SINGLE_RANGE 0x4effffff  /* 1.fffffe * 2^30 */
#define N_INT_DOUBLE_RANGE 0xc1e0000000200000 & 0x7fffffffffffffff
#define P_INT_DOUBLE_RANGE 0x41e0000000000000

void FTRC(int m,int *FPUL)
{
     pc += 2;
     clear_cause();
     if(FPSCR.PR==0){
          case(ftrc_single_ type_of(m)){
          NORM:     *FPUL = FR[m];  break;
          PINF:     ftrc_invalid(0, *FPUL);break;
          NINF:     ftrc_invalid(1, *FPUL);break;
          }
     }
     else{                  /* case FPSCR.PR=1 */
          case(ftrc_double_type_of(m)){
          NORM:     *FPUL = DR[m>>1]; break;
          PINF:     ftrc_invalid(0, *FPUL);break;
          NINF:     ftrc_invalid(1, *FPUL);break;
          }
     }
}
int ftrc_single_type_of(int m)
{
      if(sign_of(m) == 0){
         if(FR_HEX[m] > 0x7f800000)    return(NINF);   /* NaN */
         else if(FR_HEX[m] > P_INT_SINGLE_RANGE)
                   return(PINF);   /* out of range,+INF */
         else      return(NORM);   /* +0,+NORM          */
```

```
            } else {
               if((FR_HEX[m] & 0x7fffffff) > N_INT_SINGLE_RANGE)
                        return(NINF);   /* out of range ,+INF,NaN*/
               else     return(NORM);  /* -0,-NORM */
            }
      }
      int ftrc_double_type_of(int m)
      {
            if(sign_of(m) ==
               if((FR_HEX[m] > 0x7ff00000) ||
                ((FR_HEX[m] == 0x7ff00000) &&
                 (FR_HEX[m+1] != 0x00000000)))   return(NINF);      /* NaN */
               else if(DR_HEX[m>>1] >= P_INT_DOUBLE_RANGE)
                        return(PINF);/* out of range,+INF */
               else     return(NORM);/* +0,+NORM          */
      } else{
               if((DR_HEX[m>>1] & 0x7fffffffffffffff) >= N_INT_DOUBLE_RANGE)
                        return(NINF);/* out of range ,+INF,NaN*/
               else     return(NORM);/* -0,-NORM              */
      }
}
void ftrc_invalid(int sign,int *FPUL)
{
      set_V();
       if((FPSCR & ENABLE_V) == 0){
            if(sign == 0)    *FPUL = 0x7fffffff;
            else             *FPUL = 0x80000000;
      }

           else fpu_exception_trap();
      }
```

FTRC 特例

FRn,DRn	NORM	+0	–0	Positive Out of Range	Negative Out of Range	+INF	– INF	qNaN	sNaN
FTRC (FRn,DRn)	TRC	0	0	Invalid +MAX	Invalid – MAX	Invalid +MAX	Invalid – MAX	Invalid – MAX	Invalid – MAX

【注】 非规格化数的值作为零处理。

(3) 有可能发生的异常

无效运算 (Invalid operation)

6.5.20 LDS LoaD to FPU System register 系统控制指令

加载到 FPU 系统寄存器

格式		操作概略	操作码	执行状态	T 位
LDS	Rm,FPUL	Rm→FPUL	0100mmmm01011010	1	—
LDS.L	@Rm+,FPUL	(Rm)→FPUL, Rm+4→Rm	0100mmmm01010110	1	—
LDS	Rm,FPSCR	Rm→FPSCR	0100mmmm01101010	1	—
LDS.L	@Rm+,FPSCR	(Rm)→FPSCR, Rm+4→Rm	0100mmmm01100110	1	—

(1) 说明

源操作数保存到 FPU 系统寄存器 FPUL、 FPSCR。

(2) 操作内容

```
#define FPSCR_MASK 0x003FFFFF

LDSFPUL(int  m,int *FPUL)     /* LDS Rm,FPUL  */
{
              *FPUL=R[m];
              PC+=2;
}
LDSMFPUL(int  m,int *FPUL)    /* LDS.L @Rm+,FPUL  */
{
              *FPUL=Read_Long(R[m]);
              R[m]+=4;
              PC+=2;
}
LDSFPSCR(int  m)              /* LDS Rm,FPSCR  */
{
              FPSCR=R[m] & FPSCR_MASK;
              PC+=2;
}
LDSMFPSCR(int  m)             /* LDS.L @Rm+,FPSCR  */
{
              FPSCR=Read_Long(R[m]) & FPSCR_MASK;
              R[m]+=4;

              PC+=2;
}
```

(3) 有可能发生的异常

地址错误

6.5.21 STS Store from FPU System register 系统控制指令

从 FPU 系统寄存器的存储

格式	操作概略	操作码	执行状态	T 位
STS FPUL,Rn	FPUL→Rn	0000nnnn01011010	1	—
STS FPSCR,Rn	FPSCR→Rn	0000nnnn01101010	1	—
STS.L FPUL,@-Rn	Rn-4→Rn, FPUL→(Rn)	0100nnnn01010010	1	—
STS.L FPSCR,@-Rn	Rn-4→Rn, FPSCR→(Rn)	0100nnnn01100010	1	—

(1) 说明

FPU 系统寄存器 FPUL、 FPSCR 保存到目的地。

(2) 操作内容

```
STSFPUL(int  n, int *FPUL)            /* STS FPUL,Rn  */
{
             R[n]= *FPUL;
             PC+=2;
}
STSMFPUL(int  n, int *FPUL)           /* STS.L FPUL,@-Rn  */
{
             R[n]-=4;
             Write_Long(R[n],*FPUL) ;
             PC+=2;
}
STSFPSCR(int  n)                      /* STS FPSCR,Rn  */
{
             R[n]=FPSCR&0x003FFFFF;
             PC+=2;
}
STSMFPSCR(int  n)                     /* STS.L FPSCR,@-Rn  */
{
             R[n]-=4;
             Write_Long(R[n],FPSCR&0x003FFFFF)
             PC+=2;
}
```

(3) 有可能发生的异常

地址错误

(4) 使用例

```
STS
Example 1:
          MOV.L   #H'12ABCDEF, R12
          LDS     R12, FPUL
          STS     FPUL, R13
                        ; After executing the STS instruction:
                        ; R13 = 12ABCDEF
 Example 2:
          STS     FPSCR, R2
                        ; After executing the STS instruction:
                        ; The current content of FPSCR is stored in register R2
STS.L
Example 1:
MOV.L#H'0C700148, R7
STS.LFPUL, @-R7
                        ; Before executing the STS.L instruction:
                        ; R7 = 0C700148
                        ; After executing the STS.L instruction:
                        ; R7 = 0C700144, and the content of FPUL is saved at memory
                        ; location 0C700144.
Example 2:
MOV.L#H'0C700154, R8
STS.LFPSCR, @-R8
                        ; After executing the STS.L instruction:
                        ; The content of FPSCR is saved at memory location
                        0C700150.
```

第 7 章 寄存器存储体

7.1 概要

SH-2A/SH2A-FPU 内置用于高速进行伴随中断处理的寄存器保存、返回的寄存器存储体。寄存器存储体的结构如图 7.1 所示。

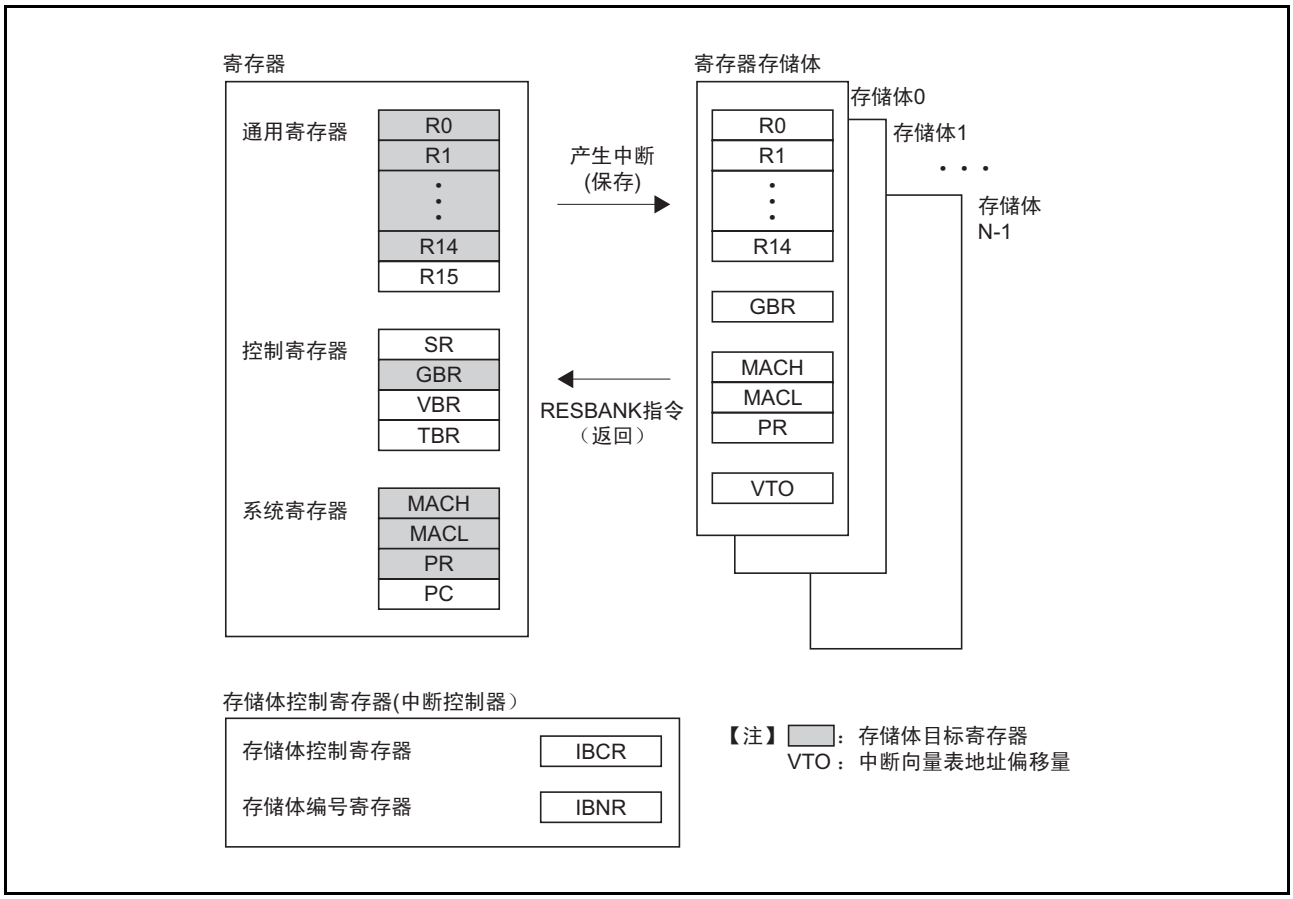


图 7.1 寄存器存储体的结构概要

7.2 寄存器存储体和存储体控制寄存器

7.2.1 存储体的目标

将通用寄存器 R0 ～ R14、全局基址寄存器 （GBR）、乘加寄存器 （MACH、 MACL）、过程寄存器（PR）和中断向量表地址偏移量 （VTO）作为存储体的目标。

7.2.2 寄存器存储体

寄存器存储体拥有从存储体 0 到存储体 N-1 的 N 个存储体 （最多 512 个存储体）。寄存器存储体为先进后出 （FILO）式堆栈，保存从存储体 0 开始按顺序进行，返回从最后保存的存储体开始。因产品不同，存储体个数 N 也不同。详细内容请参照硬件手册的 “寄存器存储体”。

7.2.3 存储体控制寄存器

(1) 存储体控制寄存器 IBCR

为对中断的优先级或中断源，设定允许 / 禁止使用寄存器存储体的寄存器。因产品不同，寄存器的规格、初始值也不同。详细内容请参照硬件手册的 “中断控制器”。

这里所示为分别对中断优先级 1 ～ 15，设定允许 / 禁止使用寄存器存储体的 IBCR 例：

IBCR （16 位，初始值 =H'0000）

bit：	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	E15	E14	E13	E12	E11	E10	E9	E8	E7	E6	E5	E4	E3	E2	E1	—

bit15 ～ 1：E15 ～ E1

对中断优先级 15 ～ 1，设定允许 / 禁止使用寄存器存储体。

bit15 ～ 1	说 明
E15 ～ E1	
0	禁止使用寄存器存储体。（初始值）
1	允许使用寄存器存储体。

bit0：保留位

读取时只读 0，写入值也只为 0。

(2) 存储体编码寄存器 IBNR （16 位，初始值 =H'0000）

bit：	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	BE1	BE0	BOVE	—	—	—	—	—	—	—	—	—	BN3	BN2	BN1	BN0

存储体编码寄存器 （IBNR）设定允许 / 禁止使用寄存器存储体，以及允许 / 禁止寄存器存储体上溢异常。另外，通过 BN3 ～ 0 指示下一个保存的存储体编码。 IBNR 由上电复位初始化为 H'0000。

bit15、 14：BE1、 BE0

设定允许 / 禁止使用寄存器存储体。

bit15、 14	说 明
BE1、 BE0	
00	在所有的中断中禁止使用存储体。忽视 IBCR 的设定。（初始值）
01	NMI、 UBC 以外的所有中断中允许使用存储体。忽视 IBCR 的设定。
10	保留 （请勿设定）
11	按照 IBCR 的设定使用寄存器存储体。

bit13：BOVE

设定允许 / 禁止寄存器存储体上溢异常。

bit13	说 明
BOVE	
0	禁止发生寄存器存储体上溢异常。（初始值）
1	允许发生寄存器存储体上溢异常。

bit12 ～ 4：保留位

读取时只读 0，写入值也只为 0。

bit3 ～ 0：BN3 ～ BN0

指示下一个保存的寄存器编码。接受使用寄存器存储体的中断时，保存至 BN3 ～ 0 所示寄存器存储体，并给 BN+1。通过执行寄存器存储体返回指令，给 BN–1 后，从寄存器存储体返回。

仅可读，不可写。

7.3 存储体保存、返回的运行

7.3.1 向存储体的保存

向寄存器存储体的保存运行如图 7.2 所示。产生中断、通过 IBCR 允许使用 CPU 接受的中断寄存器存储体时，运行如下：

(a) 设中断发生前的存储体编码寄存器IBNR的存储体编码BN的值为i。

(b) 将寄存器R0～R14、GBR、MACH、MACL、PR与接受的中断的向量表地址偏移量 （VTO）保存到BN所示的存储体i。

(c) BN的值+1。

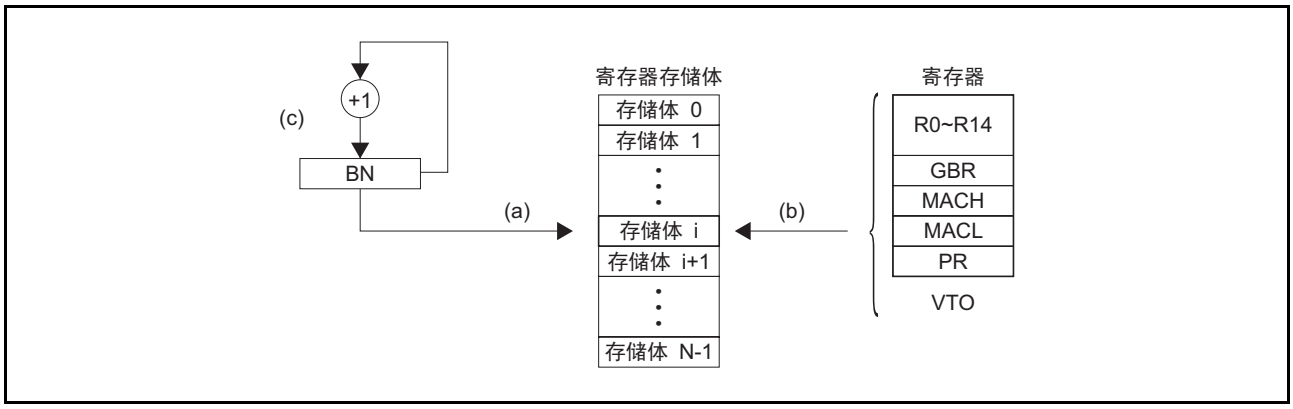


图 7.2 存储体保存的运行

寄存器存储体保存的时序如图 7.3 所示。从中断异常处理开始到开始取异常服务程序的起始指令期间进行寄存器存储体的保存。

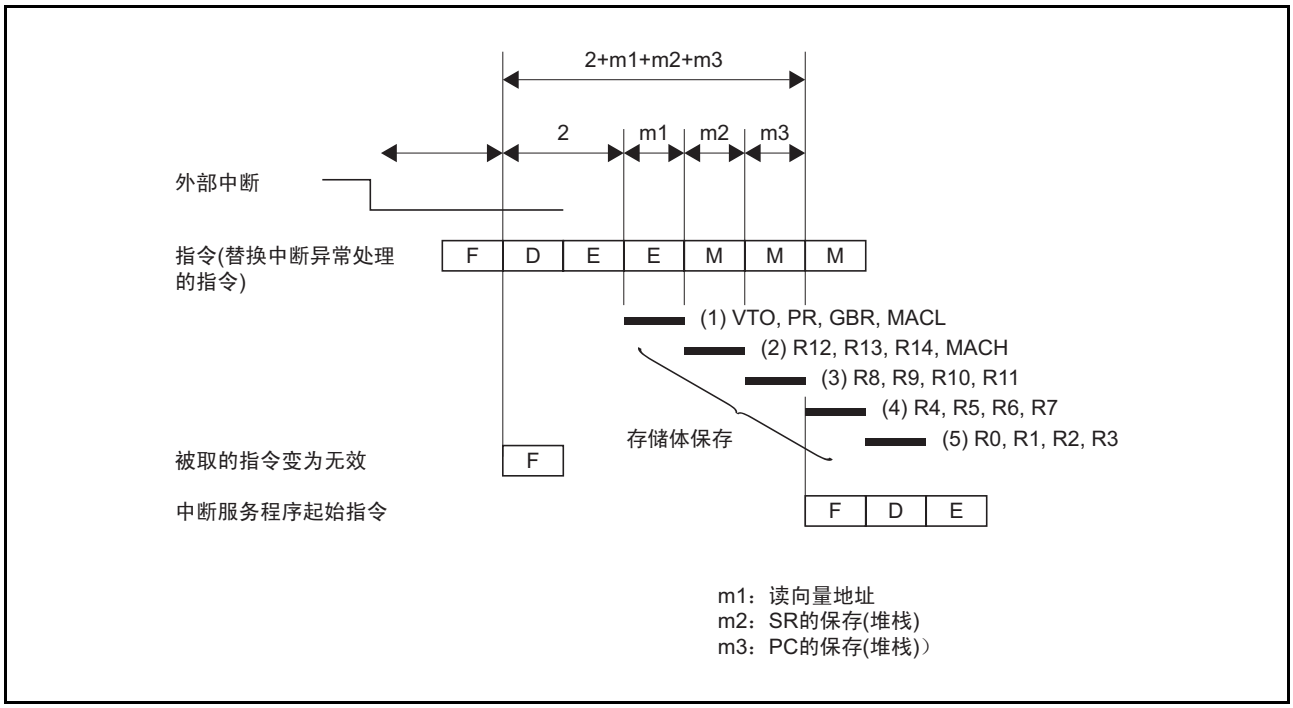


图 7.3 存储体保存的时序

7.3.2 从存储体的返回

返回保存到存储体的数据时，使用存储体返回指令 RESBANK。在中断服务程序的末尾，通过 RESBANK 指令进行存储体返回后，请通过 RTE 指令进行从异常处理的返回。

7.3.3 在已保存至所有存储体的状态下的保存、返回

在保存到寄存器存储体的所有存储体的状态下，产生中断，且 CPU 接受的中断在允许使用寄存器存储体时，可代替寄存器存储体自动向堆栈保存。可通过使用中断控制器屏蔽寄存器存储体上溢异常实现。如果发生寄存器存储体上溢异常，就不进行向堆栈的保存。详细内容请参照硬件手册的 “中断控制器”。向堆栈的自动保存、返回运行如下：

(1) 向堆栈的保存

(a) 中断异常处理时，将状态寄存器(SR)、程序计数器(PC)保存到堆栈。
(b) 将存储体目标寄存器(R0～R14、GBR、MACH、MACL、PR)保存到堆栈。保存到堆栈的寄存器顺序为MACL、 MACH、 GBR、 PR、 R14、 R13、・・・・、 R1、 R0。
(c) 将SR的寄存器存储体上溢位置1。
(d) 存储体编码寄存器IBNR的存储体编码BN保持最大值N不变。

(2) 从堆栈的返回

在 SR 的寄存器存储体上溢位置 1 的状态下，执行存储体返回指令 RESBANK 时，如下运行：

(a) 将存储体目标寄存器 （R0～R14、 GBR、 MACH、 MACL、 PR）从堆栈返回。
从堆栈返回的寄存器顺序为R0、 R1、・・・・、 R13、 R14、 PR、 GBR、 MACH、 MACL。
(b) 存储体编码寄存器IBNR的存储体编码BN保持最大值N不变。

7.4 寄存器存储体数据传送指令

为了调试的需要，具有在与通用寄存器 R0 之间传送寄存器存储体任意数据的 LDBANK 指令以及 STBANK 指令。

7.4.1 指令的说明

(1) LDBANK （从寄存器存储体向 R0 的加载）

[格式]LDBANK @Rm,R0

[运行] 向 R0 传送以 Rm 所示寄存器存储体地址开始的 4B 数据。

(2) STBANK （从 R0 向寄存器存储体的存储）

[格式]STBANK R0,@Rn

[运行] 向 Rn 所示寄存器存储体地址传送 R0。

7.4.2 寄存器存储体的寻址

寄存器存储体传送指令的地址 （LDBANK 为 Rm 的值； STBANK 为 Rn 的值）与寄存器存储体的入口的对应如图 7.4 所示。通过地址的 15 ～ 7 位 （BN）指定存储体编码，通过地址的 6 ～ 2 位 （EN）指定存储体内的入口 （R0 ～ R14、 GBR、 MACH、 MACL、 PR、 VTO）。将地址的 31 ～ 16 位和 1 ～ 0 位全部清 0。如果地址的 31 ～ 16 位和 1 ～ 0 位不全为 0 时、通过地址的 15 ～ 7 位指定不存在的存储体时、或在地址的 6 ～ 2 位指定不存在的入口时，不保证运行。

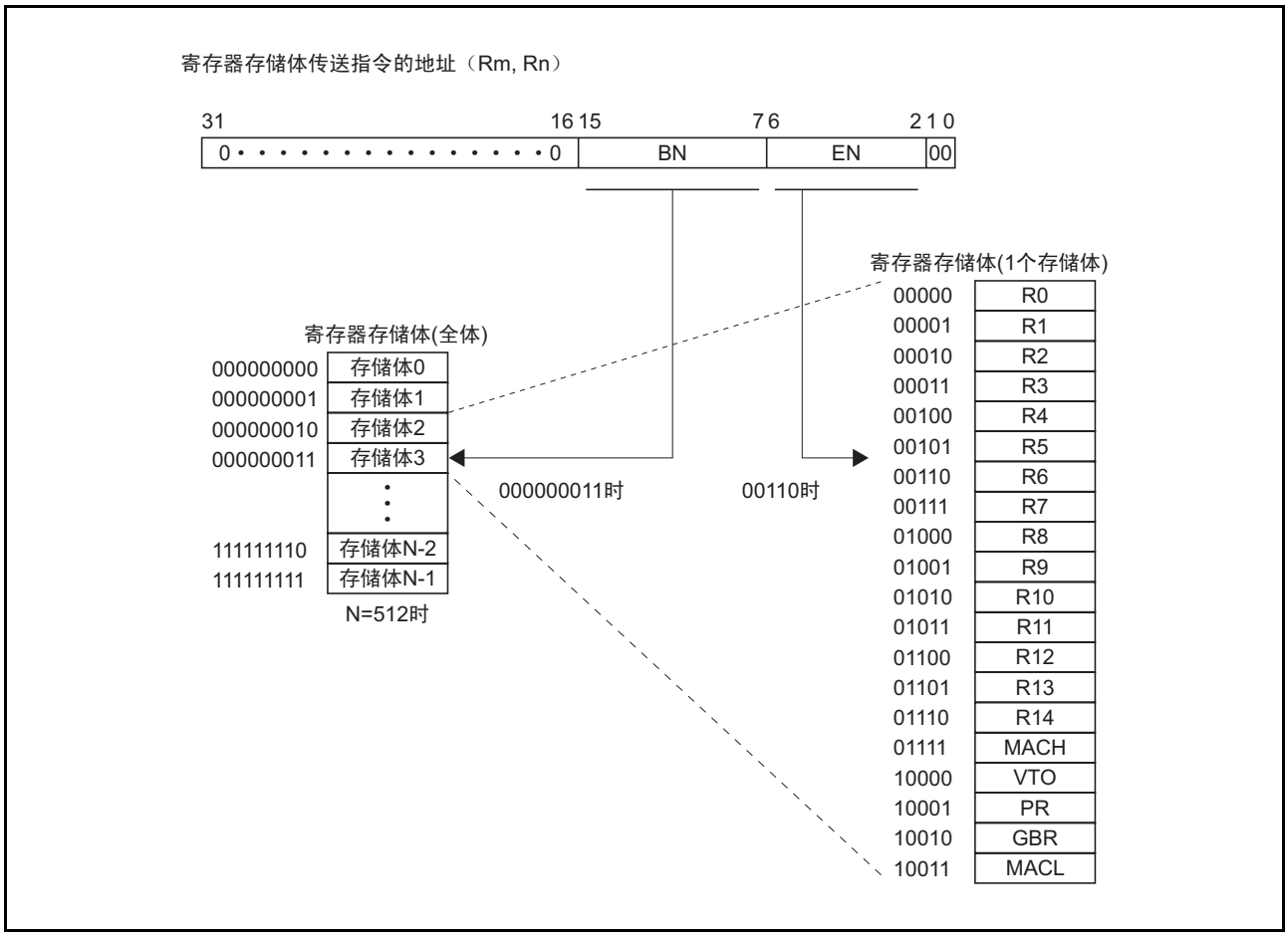


图 7.4 寄存器存储体的寻址

7.5 寄存器存储体的异常

寄存器存储体的异常 （寄存器存储体错误）有寄存器存储体上溢和寄存器存储体下溢 2 种。

7.5.1 寄存器存储体错误发生源

(1) 寄存器存储体上溢

在保存到寄存器存储体的所有存储体的状态下，产生中断，且 CPU 接受的中断在允许使用寄存器存储体时，不能通过中断控制器屏蔽寄存器存储体上溢异常的情况下就发生此异常。此时，存储体编码寄存器 IBNR 的存储体编码 BN 保持存储体数 N 不变，不可向寄存器存储体保存。

(2) 寄存器存储体下溢

在完全未保存到寄存器存储体的状态下，执行 RESBANK 指令时发生此异常。此时， R0 ～ R14、 GBR、 MACH、 MACL、 PR 的值不变。存储体编码寄存器 IBNR 的存储体编码 BN 保持 0 不变。

7.5.2 寄存器存储体错误异常处理

发生寄存器存储体错误时，开始寄存器存储体错误异常处理。此时， CPU 进行如下运行：

(1) 将状态寄存器 （SR）保存到堆栈。

(2) 将程序计数器 （PC）保存到堆栈。寄存器存储体上溢时保存的 PC 值为最后执行指令的下一条指令的起始地址。寄存器存储体下溢时保存的 PC 值为该 RESBANK 指令的起始地址。
存储体上溢时为了防止多重中断，将成为存储体上溢源的中断级写入状态寄存器 （SR）的中断屏蔽位 （I3 ～ I0）。

(3) 从发生的寄存器存储体错误对应的异常处理向量表取出异常服务程序的起始地址，然后从此地址开始执行程序。

7.6 SR 的寄存器存储体上溢位 （BO 位）

可通过由 RTE 指令产生的 SR 的返回来改写 BO 位。即使执行 RESBANK 指令也不可改写 BO 位。通过中断时的存储体上溢，将允许中断控制器内的异常发生设置为 OFF 时， BO 位置 1。通过中断时的存储体上溢，将允许中断控制器内的异常发生设置为 ON 时，不能改写 BO 位。通过 LDC Rm,SR,LDC.L@Rm+,SR 指令改写 BO 位。

第 8 章 流水线运行

本章说明各指令的流水线运行。这是用于计算 CPU 指令执行状态数 （系统时钟周期数）的信息。

SH-2A/SH2A-FPU 为 2 条指令并行型 （2-ILP, Instruction-Level-Parallelism）的超标量流水线处理微处理器。指令执行实现流水线化，可并行执行 2 条指令。采用哈佛体系结构，存储器存取与取指令相互不竞争。并且，因为有取指令单元，所以即使在取指令中 CPU 内核也不停止。

8.1 流水线的基本结构

SH-2A/SH2A-FPU 中具有以下流水线。（图 8.1）

- 整数流水线1、 2 ：处理整数运算。
- 存储器存取流水线 ：处理存储器存取和向FPU的数据加载。
- 乘法器流水线 ：处理乘法指令和来自FPU的数据存储。
- 转移流水线 ：处理转移指令。
- 移位流水线 ：处理移位指令。
- FPU加载存储流水线 ：处理FPU的加载储存指令。
- FPU算术运算流水线 ：处理FPU的算术运算。
- FPU除法平方根运算流水线 ：处理FPU的除法与平方根运算。

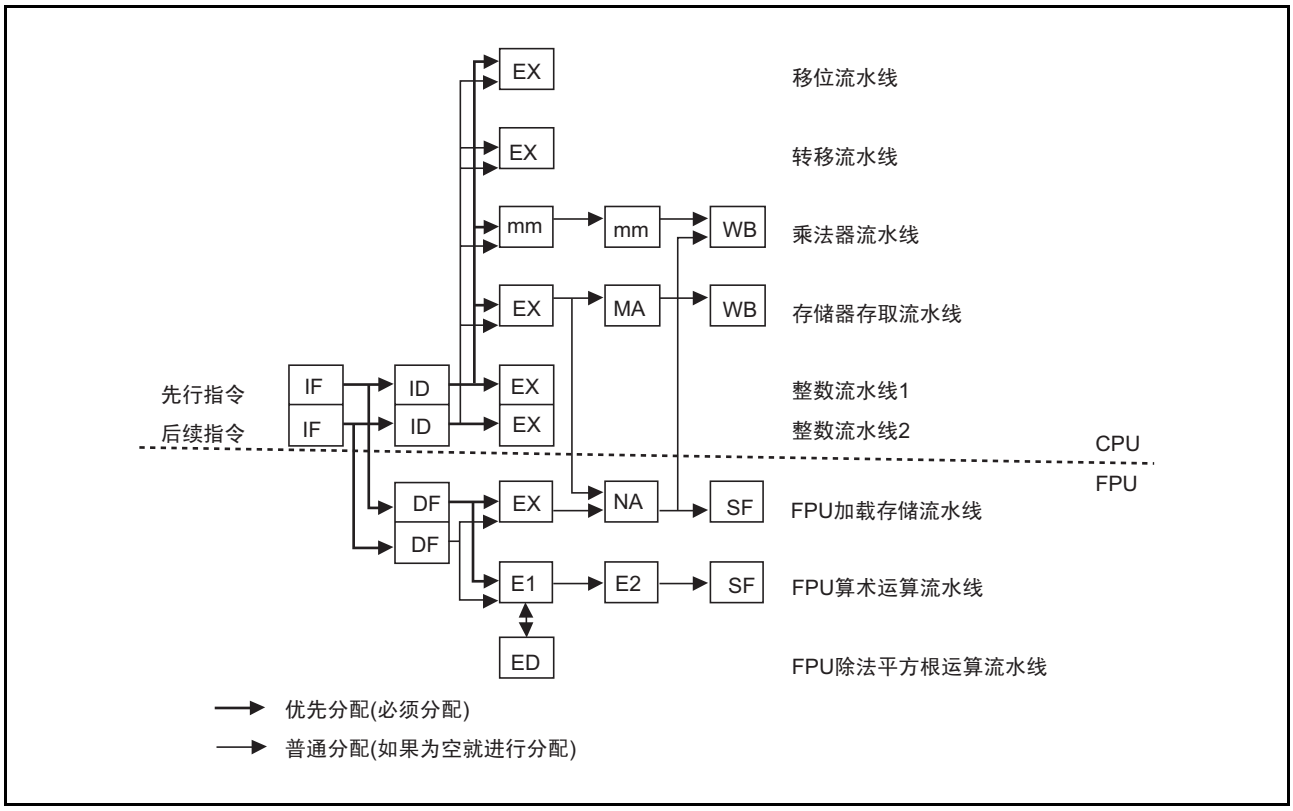


图 8.1 SH-2A/SH2A-FPU 的流水线

所有的指令首先通过整数流水线处理，如果有必要也可交给其他流水线。这些流水线均可独立运行。因此，如果没有竞争，总是连续发行 2 条指令。

进行存储器存取的指令以及由 CPU 向 FPU 加载数据的指令，使用存储器存取流水线。

乘法指令、乘法结果寄存器存取指令使用乘法器流水线。存储来自 FPU 的数据的指令使用乘法器流水线的 WB 阶段。

转移指令使用转移流水线。

移位指令使用移位流水线。

FPU 内部的寄存器的移动以及从 FPU 向存储器、 CPU 的数据交换的指令，使用 FPU 加载存储流水线。

进行 FPU 算术运算的指令，使用 FPU 算术运算流水线。

在 FPU 算术运算中，关于 FDIV、 FSQRT，使用 FPU 算术运算流水线与 FPU 除法平方根运算流水线。

详细内容参照 “8.9 各指令的流水线运行”。

各 CPU 流水线的阶段的详细内容说明如下：

- IF ：取指令 从保存程序的存储器中取指令。
- ID ：指令解码 解读所取的指令。
- EX ：指令执行 根据解读的结果进行数据运算和地址计算。
- MA ：存储器存取 进行存储器的数据存取。
- 通过存储器存取的指令以及从CPU向FPU加载数据的指令来产生。
- mm ：乘法器存取 进行乘法器的存取。通过乘法指令或对乘法结果寄存器的存取的指令来产生。
- WB ：回写 将存储器存取、乘法器存取的结果 （数据）返回到寄存 器。

各 FPU 流水线阶段的详细内容说明如下：CPU 和 FPU 的流水线共享第 1 阶段的取指令 （IF）。

- DF ：FPU解码 解读所取的指令。
- E1 ：FPU执行的第1阶段 对浮点运算进行初始化。
- E2 ：FPU执行的第2阶段 进行浮点运算。
- SF ：FPU存储 完成浮点运算，将结果写入FPU 寄存器。
- ED ：FPU除法平方根运算 仅用于FDIV、 FSQRT。
- EX ：FPU加载存储第1阶段 进行浮点加载存储指令的数据准备。
- NA ：FPU加载存储第2阶段 进行浮点加载存储指令的数据交换。

ID、 DF 以后的各阶段长度均相同。仅 IF 有时会因等待数据而延长 , 但是由于取指令单元和流水线独立运行，因此即使在此情况下，已经完成取的指令的流水线仍继续运行。

如图 8.2 所示，指令的各阶段随着指令执行流程而构成流水线，流水线的基本流程如图 8.2 所示。将执行 1 个阶段的期间称为槽，用 “←→”表示。每条指令至少由 3 个阶段构成。

所有指令均有 IF、 ID 和 EX 的 3 个阶段 （整数流水线）。此后，需要的流水线也同时运行，并进行指令的处理。

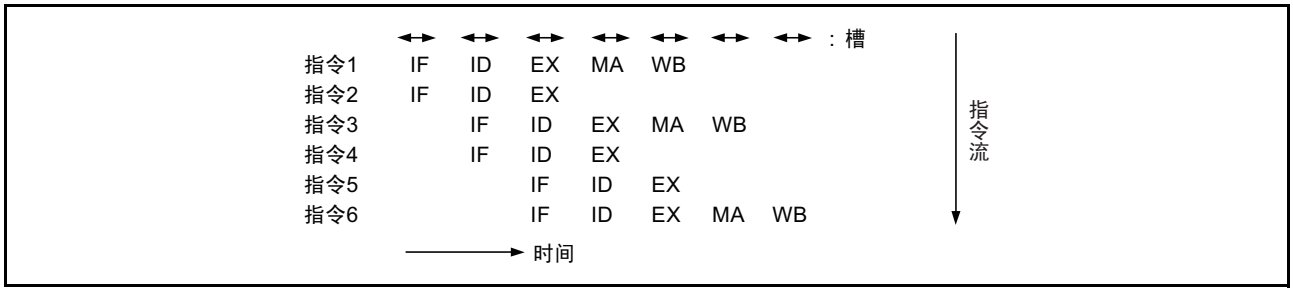


图 8.2 流水线的基本结构

8.2 槽和流水线的流程

1 个阶段的执行期间称为槽。有关槽的规则如下所示：

(1) 必须在1个槽内执行指令的各阶段（IF、ID、EX、MA、WB、mm、E1、E2、DF、ED、SF、NA），而不能在1个槽内执行2个或2个以上的阶段 （图8.3）。并且， ED阶段的运行与槽无关。

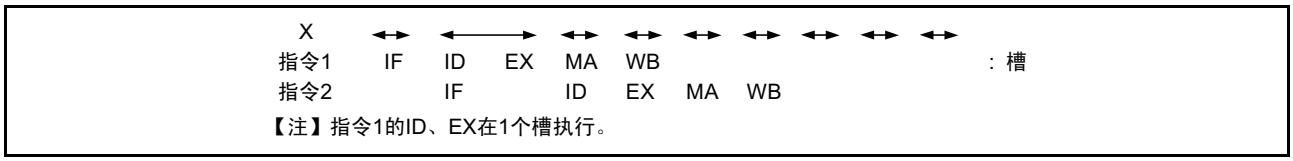


图 8.3 不可能的流水线流程 （1）

(2) 在1个槽内其他指令的不同阶段在整数流水线时最多可设定2个，在其他的流水线时最多只能设定1个。不能同时执行大于这些数量的阶段。（图8.4）。

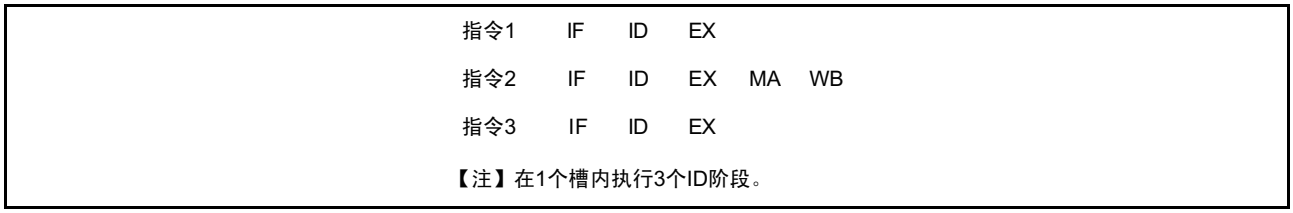


图 8.4 不可能的流水线流程 （2）

(3) 执行1个槽所需的状态数 （系统时钟周期数） S通过以下条件算出：

(a) S= （1个槽内所含的各指令阶段中的最多状态数）
即：因最长阶段引起其他短阶段的指令停止。

(b) 各阶段的执行状态数……

- IF ：用于取指令的存储器存取时钟数
（因为有取缓冲器并进行预先取指令，所以除需要立即对取的指令进行解码的情况外，不停止流水线。）
- ID ：总是1个状态
- EX ：总是1个状态
- MA ：用于数据存取的存储器存取时钟数
- WB ：总是1个状态
- mm ：总是1个状态
- DF ：总是1个状态
- E1 ：总是1个状态
- E2 ：总是1个状态
- SF ：总是1个状态
- ED ：总是1个状态，但是运行与槽无关。
- NA ：总是1个状态

例如：指令1、 2的IF （用于取指令的存储器存取）需要2个周期、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个周期的流水线流程如图8.5所示。“－”表示停止。另外，该图中为了简单起见，没有考虑超标量。

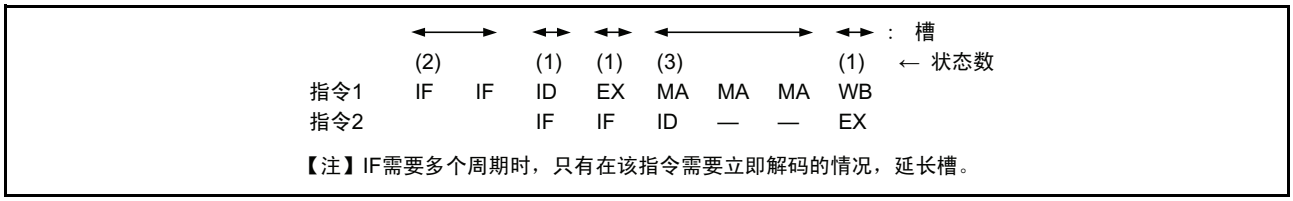


图 8.5 需要多个周期的槽

8.3 指令执行、并行执行性

SH-2A/SH2A-FPU 为 2 条指令并行型 （2-ILP, Instruction-Level-Parallelism）的超标量流水线处理微处理器。 2 条指令进入 ID 阶段时可同时执行 2 条指令。（图 8.6）

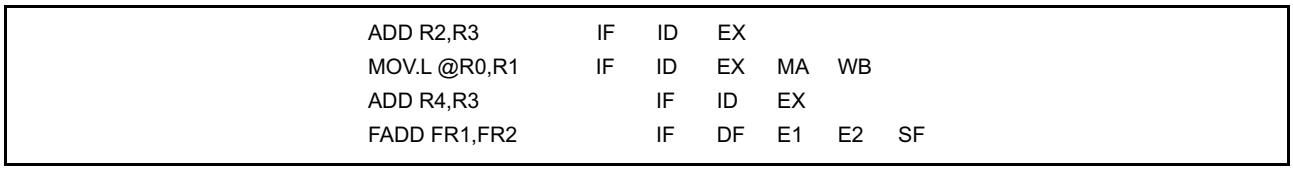


图 8.6 并行执行例

但是，以下情况不能并行执行：

- 发生资源竞争时 （参照8.3.1说明）
- 等待已发行指令的结果时 （参照8.3.2说明）
- 发生寄存器竞争/标志竞争时 （参照8.3.3说明）
- 多周期指令作为先行指令执行时 （参照8.3.4说明）
- 32位指令作为先行指令执行时 （参照8.3.5说明）
- 使用FPSCR的指令、 FPU指令及FPU相关的CPU指令 （参照8.3.6说明）
- 引起转移的带延迟的无条件转移指令、延迟槽 （参照8.3.7说明）

当这些竞争不发生并且 2 条指令的 IF 完成时， SH-2A/SH2A-FPU 可并行执行 2 条指令。
在以下项目中分别进行说明。说明时使用以下的称呼：

- 先行指令…同一槽内，前方的指令
- 后续指令…同一槽内，后方的指令
- 已发行指令… 已经发行的指令的总称

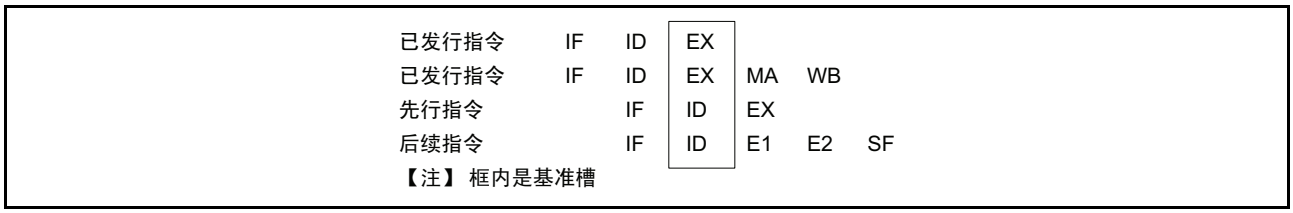


图 8.7 先行、后续、已发行指令的定义

8.3.1 资源竞争的详细内容

由于整数流水线以外的流水线仅各有 1 条，所以当先行指令和后续指令要同时使用时发生竞争，后续指令等待被执行。竞争发生时情况如下：

(1) 先行指令和后续指令均执行存储器存取的指令时。（图8.8）
或CPU→FPU的数据传送指令和存储器写指令的组合。（图8.9）
或均为CPU→FPU的数据传送指令的组合。
此时，发生存储器存取流水线的竞争。

```
MOV.L   @R1+,R2   IF   ID   EX   MA
MOV.L   @R1+,R3   IF   --   ID   EX   MA
【注】 每个槽中最多为1个存储器存取(MA)。
```

图 8.8 存储器存取竞争例

```
LDS     R0,FPUL   IF   ID   EX                  : CPU流水线
                  IF   DF   EX   NA   SF        : FPU流水线
MOV.L   R1,@R3    IF   --   ID   EX   MA        : CPU流水线
【注】LDS指令与存储器写指令竞争。
```

图 8.9 LDS 指令和存储器写指令竞争例

而且， FPU→CPU的数据传送指令与存储器存取指令不发生竞争。（图8.10）
另外， CPU→FPU的数据传送指令与存储器读指令也不发生竞争。（图8.11）

```
STS     FPUL,R0   IF   ID   EX   WB             : CPU流水线
                  IF   DF   EX   NA   SF        : FPU流水线
MOV.L   @R1+,R3   IF   ID   EX   MA   WB   WB   : CPU流水线
【注】STS指令与存储器存取指令不竞争。
```

图 8.10 STS 指令和存储器存取指令例

```
LDS     R0,FPUL   IF   ID   EX                  : CPU流水线
                  IF   DF   EX   NA   SF        : FPU流水线
MOV.L   @R1+,R3   IF   ID   EX   MA   WB        : CPU流水线
【注】LDS指令和存储器读指令不竞争。
```

图 8.11 LDS 指令和存储器读指令例

(2) 先行指令和后续指令均为使用乘法器的指令时。（图8.12）
但是即使在锁上已发行指令时也会发生乘法器竞争。（图8.13）
另外，因为MACH、MACL的读取指令、MULR指令、将FPUL和FPSCR值传送到CPU的指令共享读取总线，所以会发生竞争。（图8.14）

```
MULS.W  R2,R1     IF  ID  mm  mm
MULR    R0,R3     IF  --  ID  mm  mm  mm  WB
```

图 8.12 乘法器竞争例

```
乘法器锁上                    ←→
LDS.L   @R1+,MACH  IF  ID  EX  MA  WB
MULR    R0,R3      IF  --  --  ID  mm  mm  mm  WB
```

图 8.13 由于已发行指令而发生竞争例

```
STS   MACH,R0   IF  ID  EX  MA  WB
STS   FPUL,R1   IF  --  ID  mm  mm  mm  WB
【注】使用乘法结果读取总线的指令相互竞争。
```

图 8.14 使用乘法结果读取总线指令的竞争

(3) 先行指令和后续指令均为移位指令或者循环指令时。（图8.15）

```
SHAD R0,R1   IF  ID  EX
SHAD R2,R3   IF  --  ID  EX
```

图 8.15 在移位指令情况下的竞争例

(4) 先行指令和后续指令均为FPU算术运算指令时。（图8.16）
另外，关于FPU算术运算指令，双精度时以及在FDIV、 FSQRT指令的情况下会发生复杂的资源竞争。详细内容请参照 “8.6 由FPU引起的竞争”。

```
FADD FR0,FR1   IF  DF  E1  E2  SF
FADD FR2,FR3   IF  --  DF  E1  E2  SF
```

图 8.16 在 FPU 算术运算指令情况下的竞争例

(5) 先行指令和后续指令均为FPU加载存储指令时。（图8.17）

```
FNEG FR0     IF  DF  EX  NA  SF
FMOV FR1,FR3 IF  --  DF  EX  NA  SF
```

图 8.17 在 FPU 加载存储指令情况下的竞争例

8.3.2 由等待已发行指令结果引起的竞争的详细内容

已发行指令的结果作为源使用时，在等待该指令的等待时间之后再执行。这时有以下情况：

- 等待存储器存取的结果时 （详细内容请参照 “8.5 存储器加载指令对流水线的影响”）
- 等待FPU运算结果时 （详细内容请参照 “8.6 由FPU引起的竞争”）
- 等待乘法结果时 （详细内容请参照 “8.7 由乘法器引起的竞争）

此时，先行指令引起竞争时，后续指令也需要等待执行。

后续指令引起竞争时，如果没有另外的竞争就执行先行指令。

8.3.3 寄存器竞争 / 标志竞争的详细内容

以下情况时，在同一槽内发生寄存器竞争 / 标志竞争。

(1) 后续指令将先行指令的目标寄存器、标志作为源寄存器、标志使用时。（但是，除先行指令是0个等待时间的指令的情况）（图8.18、图8.19）

```
CMP/EQ R2,R3    IF  ID  EX
BF              IF  --  ID  EX
```

图 8.18 在先行目标、后续源情况下的标志竞争例

```
MOV  R3,R4      IF  ID  EX
ADD  R4,R5      IF  ID  EX
```

图 8.19 在 0 个等待时间指令与后续指令情况下的不竞争例

(2) 后续指令对先行指令的目标寄存器、标志进行写入时。（但是，只有在通过乘法指令、除法指令、LDBANK指令、 RESBANK指令、 MOVMU指令、 MOVML指令以外的指令对FPU寄存器、 CS位以外的寄存器、标志进行写入时才发生竞争。在乘法指令、除法指令、 LDBANK指令、 RESBANK指令中检测不出该竞争。在MOVMU指令中仅对Rn检测出竞争，在MOVML指令中仅对R0检测出竞争。在这些指令中对这些以外的寄存器，不发生此种竞争。）（图8.20～图8.25）

```
ADD R3,R4     IF   ID   EX
MOV R5,R4     IF   --   ID   EX
```

图 8.20 对先行指令的目标进行盖写的指令发生竞争的例 1

```
MOV.L  @R0,R1   IF   ID   EX   MA
MOV.L  @R2,R1   IF   --   --   ID   EX
```

图 8.21 对先行指令的目标进行盖写的指令发生竞争的例 2

```
CLIPS.B  R3        IF   ID   EX
CLIPS.B  R4        IF   ID   EX
```

图 8.22 CS 位不竞争例

```
MOV    R5,R6     IF   ID   EX
MULR   R0,R6     IF   ID   mm   mm   mm   WB
```

图 8.23 MULR 不竞争例

```
MOV      R5,R6        IF   ID   EX
MOVMU.L  @R15+,R13    IF   ID   EX   MA   MA   MA   WB
```

图 8.24 MOVMU.L 不竞争例

```
MOV      R5,R13       IF   ID   EX
MOVMU.L  @R15+,R13    IF   --   ID   EX   MA   MA   MA   WB
```

图 8.25 MOVMU.L 竞争例

8.3.4 由多周期指令引起的竞争的详细内容

将执行状态不是 1 的指令称为 “多周期指令”。该指令中有以下规则：

(1) 多周期指令作为先行指令执行时，不能与后续指令并行执行。

(2) 多周期指令的执行中，不是最后的槽时，不能重新执行下一条指令。这里，所谓执行中是指从指令的ID阶段开始计算，小于执行状态周期数的槽。

(3) 在多周期指令的执行状态的末尾 （最终槽：正在执行的状态的周期数），能与下一条指令并行执行。下一条指令即使是32位指令也能并行执行。

(4) 多周期指令能与先行指令的单一周期指令 （执行状态=1条指令）并行执行。
此时的例如图8.26所示。

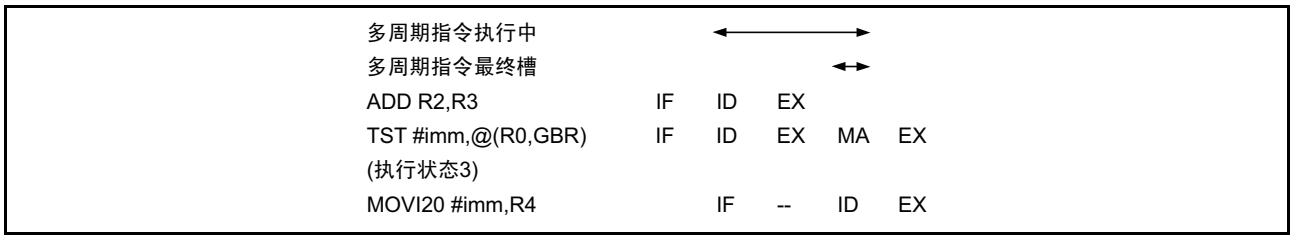


图 8.26 多周期指令的执行例

(5) 当后续指令是BAND.B、BANDNOT.B、BLD.B、BLDNOT.B、BOR.B、BORNOT.B、BXOR指令时，32位指令的多周期指令的BAND.B、 BANDNOT.B、 BLD.B、 BLDNOT.B、 BOR.B、 BORNOT.B、 BXOR指令能与后续指令并行执行。（图8.27）

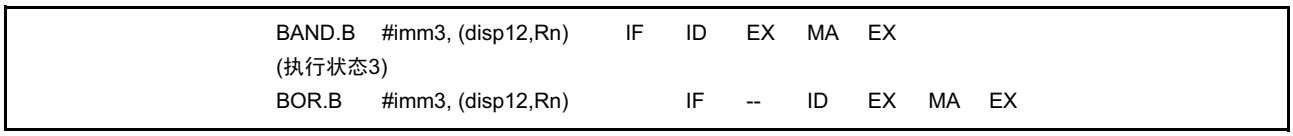


图 8.27 32 位的位操作指令连续时的执行例

(6) 除 （5）的情况外32位多周期指令不能与后续指令并行执行。（图8.28）

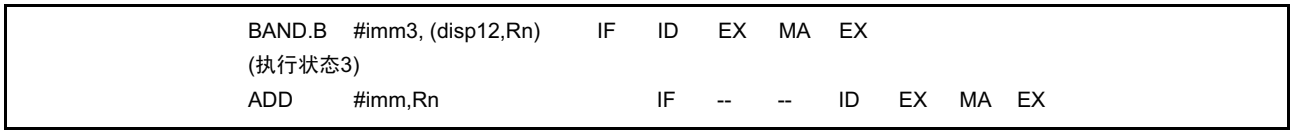


图 8.28 32 位多周期指令的执行例

8.3.5 由 32 位指令引起的竞争的详细内容

32 位指令的执行有以下规则：

(1) 先行指令为 32 位指令时，不可并行执行。（图 8.29）

(2) 后续指令为 32 位指令时，先行指令可执行，后续指令不可执行。（图 8.29）

(3) 多周期指令的最终槽与 32 位指令可并行执行。（图 8.26）

(4) 先行指令为 32 位的多周期指令时， BAND.B、 BANDNOT.B、 BLD.B、 BLDNOT.B、 BOR.B、 BORNOT.B、 BXOR 指令的最终槽与后续指令 （仅限 AND.B、 BANDNOT.B、 BLD.B、 BLDNOT.B、BOR.B、BORNOT.B、BXOR 指令时）可并行执行，不过，与这些以外的指令组合时，不可并行执行。（图 8.27、图 8.28）

(5) IF 未完成时， 32 位指令的高 16 位和低 16 位不可执行。（图 8.30）

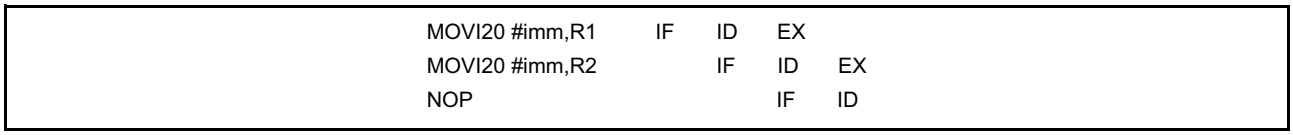


图 8.29 在 32 位指令情况下的竞争例

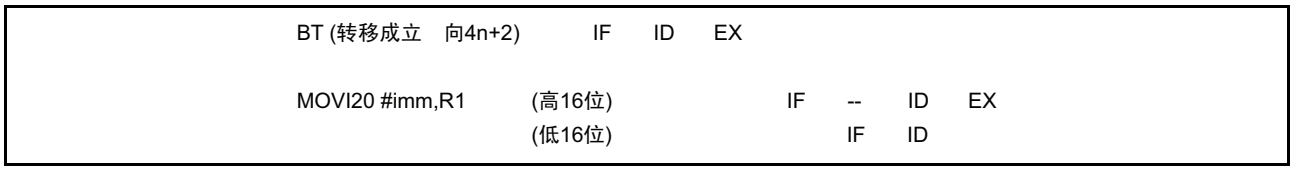


图 8.30 在 32 位指令的内部停止例

8.3.6 由使用 FPSCR 指令引起的竞争的详细内容

将使用 FPSCR 的指令作为先行指令提供时，不可与所有的指令并行执行。将该指令作为后续指令提供时，不可与 FPU 指令及 FPU 相关的 CPU 指令并行执行。（图 8.31）

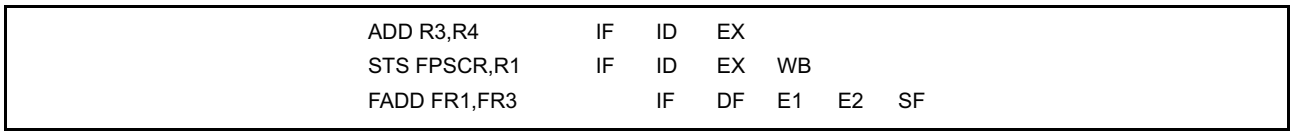


图 8.31 使用 FPSCR 的指令的竞争例

8.3.7 由转移指令引起的竞争的详细内容

由转移指令引起的竞争有以下规则：

(1) 转移指令不转移时，可并行执行。

(2) 将转移指令作为后续指令提供时，与是否转移无关，可与先行指令并行执行。

(3) 将转移指令作为先行指令提供时，如果转移发生，不可与后续指令并行执行。即使延迟槽已经完成 IF，也不可并行执行。（图8.32）

(4) 延迟槽在有转移指令的EX阶段的下一个槽执行ID。

(5) 未对延迟槽进行取时，延迟转移指令等待执行。

(6) 转移指令与转移指令竞争。

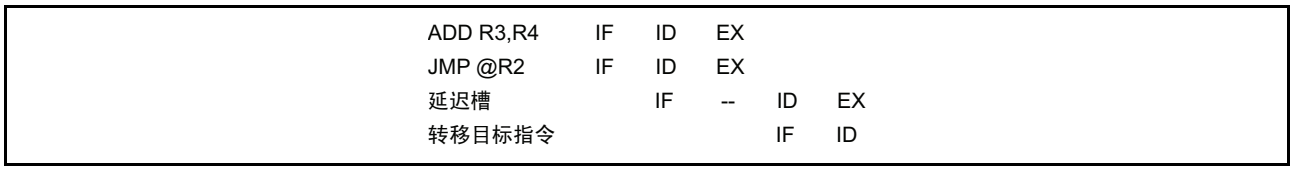


图 8.32 转移指令的竞争例

8.4 指令执行状态数

通过 EX 阶段的执行间隔数各指令的执行状态数。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。

例如：在图 8.33 所示流水线流程的情况下，因为指令 1 和指令 2 的 EX 阶段的执行间隔为 4 个阶段，所以指令 1 的执行时间为 4 个状态；因为指令 2 和指令 3 的 EX 阶段的执行间隔是 1 个阶段，所以指令 2 的执行时间为 1 个状态。

如果程序在指令 3 结束，就假设指令 3 的下一条指令是指令 4 （MOV Rm, Rn），根据指令 3 和指令 4 的 EX 阶段的执行间隔来数指令 3 的执行时间 （在图 8.33 中，指令 3 的执行时间为 1 个状态）。

图 8.33 中的指令 1 ～指令 3 的总执行时间为 4+1+1 ＝ 6 个状态。

另外，该图中为了简单起见，未考虑超标量。

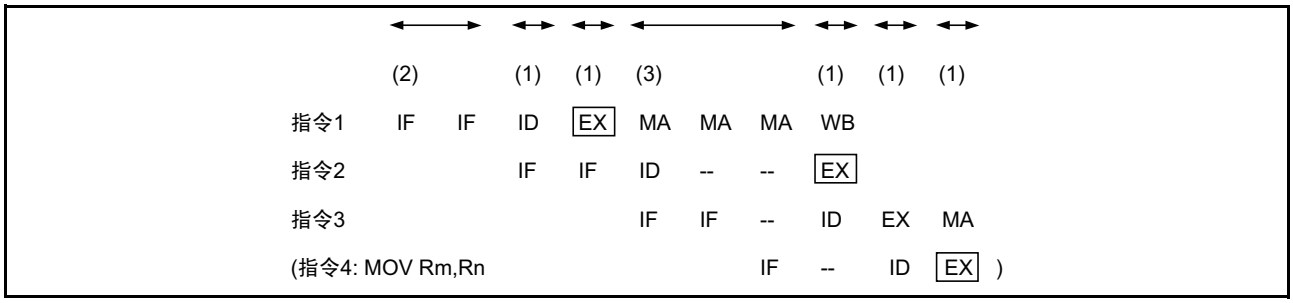


图 8.33 数指令执行状态数的方法例

8.5 存储器加载指令对流水线的影响

在流水线最后的 WB 阶段，存储器的加载指令将数据返回到目标寄存器。如果注意该加载指令 （作为加载指令 1）和紧接着的指令 （作为指令 2），发现指令 2 的 EX 阶段出现在加载指令 1 的 WB 阶段完成之前。

此时，假如指令 2 要使用加载指令 1 的目标寄存器，因为该寄存器的内容尚未准备好，所以指令 2 的 ID 的执行延迟指令 1 的等待时间。加载指令 1 的目标寄存器即使不是指令 2 的源而是指令 2 的目标寄存器，时也同样。

另外，在加载指令 1 的目标是状态寄存器 （SR）并且指令 2 取进并使用其中的标志时 （例如 ADDC 等），再停止 1 个槽。

当发生这样的寄存器竞争时，可使用该目标寄存器的槽为指令 1 的 MA 完成后的周期。此情况如图 8.34 所示。

因此，紧接加载指令之后，如果写配置使用该结果的指令的程序，执行速度就会下降。一般情况下，加载指令的等待时间为 2 个，所以使用加载结果的指令配置在加载指令的 3 ～ 4 条指令之后时，速度就不会降低。存储器存取指令作为先行指令执行时，为 4 条指令之后。作为后续指令执行时，为 3 条指令之后。

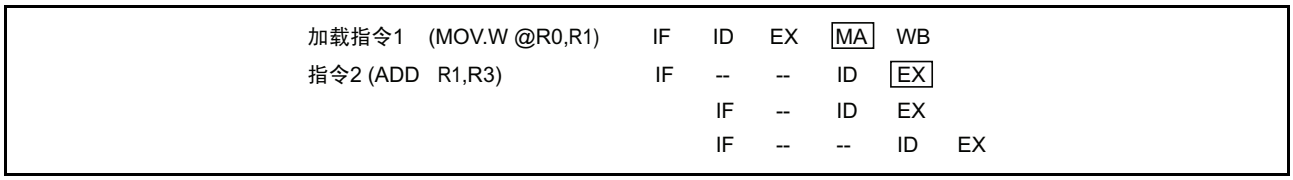


图 8.34 存储器加载指令对流水线的影响

8.6 由 FPU 引起的竞争

通过后续的浮点数算术运算指令或者 FMOV FRm,FRn 指令读 保存浮点数算术运算指令、 FMOV 指令或浮点数加载指令结果的寄存器 （FR0 ～ FR15， FPUL）（作为源寄存器使用）时，运算完成后发行下一条指令。其结果是，该指令等待之前的运算指令的等待时间周期。（图 8.35）即使在后续的指令将结果寄存器作为源时， 0 个等待时间的指令也可与后续指令并行执行。（图 8.36）

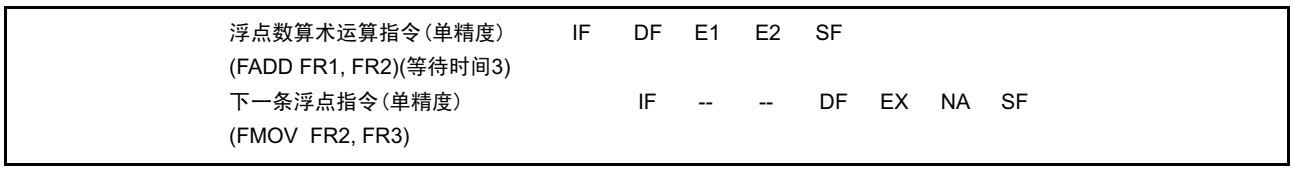


图 8.35 后续指令使用 FPU 运算结果时的例子

图 8.36 0 个等待时间的指令的结果作为源使用的例子

通过后续 FMOV， STS.L 指令读保存浮点数算术运算指令结果的寄存器 （FR0 ～ FR15）（作为源寄存器使用）时，如果向存储器输出值，等待时间就缩短 1 个周期。（图 8.37）

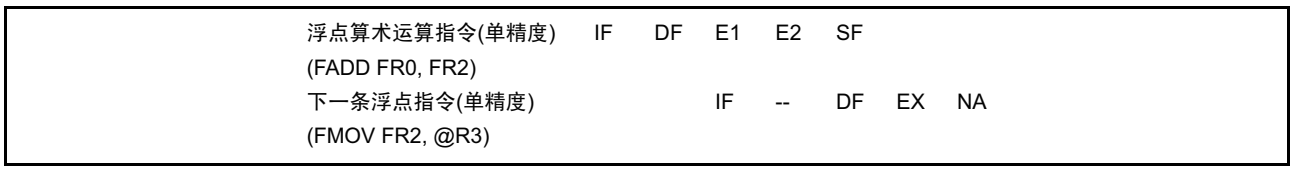


图 8.37 紧接 FPU 运算之后，其结果写入存储器时的例子

通过后续 STS 指令读保存浮点数算术运算指令结果的寄存器 （FPUL）（作为源寄存器使用）时，如果向 CPU 输出值，等待时间就缩短 2 个周期。（图 8.38）

图 8.38 紧接 FPU 运算之后，其结果传送到 CPU 时的例子

到 FCMP 指令的结果反映到 Tbit 为止，单精度需要 2 个周期，双精度需要 3 个周期。结果是，如果该指令 （后续指令）参照 T 位，就延迟执行上述槽的时间。（图 8.39）

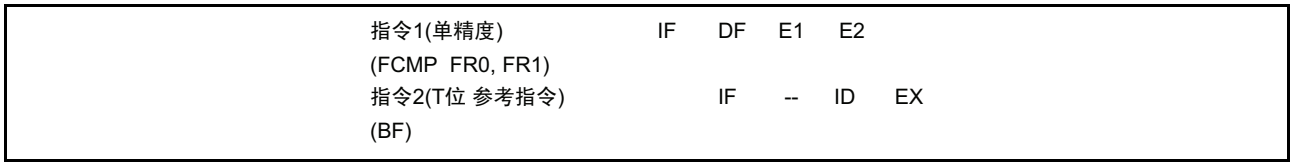


图 8.39 紧接 FCMP 指令之后，参照 T 位时的例子

如果使用 LDS 或者 LDS.L 指令更改 FPSCR 的值，后续的指令就会延迟执行 3 个槽的执行期间 （图 8.40）

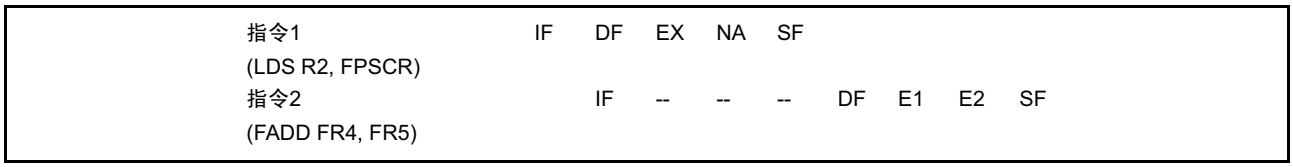


图 8.40 紧接 FPSCR 加载后，进行 FPU 运算时的例

使用 STS 或者 STS.L 指令读 FPSCR 的值时，已发行的运算完成后读 FPSCR。结果是，延迟执行先运算的等待时间 +1 个槽的时间。（图 8.41）

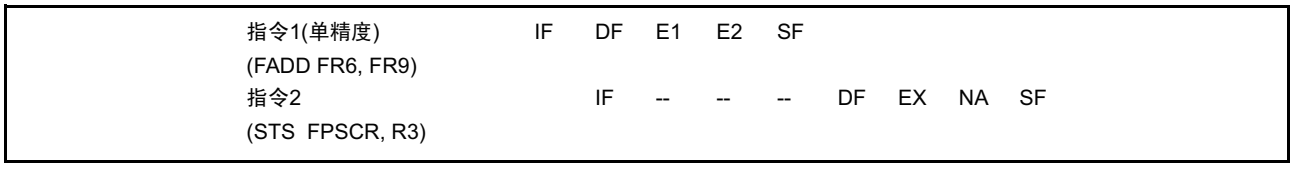


图 8.41 读 FPSCR 时的例子

双精度浮点数算术运算指令 （FADD、 FSUB、 FMUL）在 E1 阶段需要 6 个周期。在此期间，其他浮点算术运算指令不进入 E1 阶段。如果在双精度浮点数算术运算指令结束 E1 阶段前出现其他浮点算术运算指令，该浮点算术运算指令就会延迟执行所定槽的时间，并在双精度浮点数算术运算指令结束 E1 阶段后进入 E1 阶段。另外，可执行此期间后续的浮点加载存储指令。（图 8.42）

```
FADD DR4, DR6      IF   DF   E1   E1   E1   E1   E1   E1   E2   SF
FABS  DR0          IF   DF   EX   NA   SF
STS   FPUL, R0          IF   DF   EX   NA
FMUL DR2, DR0           IF   --   --   --   --   --   DF   E1   E2   SF
```

图 8.42 双精度 FPU 运算与后续 FPU 指令的运行例

FDIV、 FSQRT 指令初始化使用了 E1 阶段后，以独立的计算机 （ED 阶段）进行运算，随后，回写运算结果。这些指令之后的浮点算术运算指令，如下运行。另外，关于各指令的流水线运行的详细说明，请参照“8.9 各指令的流水线运行”。

(1) 在初始化中使用E1阶段期间，其他的浮点算术指令不进入E1阶段。FDIV、FSQRT初始化结束后，这些指令进入E1阶段。

(2) 除去使用FDIV、 FSQRT指令结果寄存器的情况，立即执行FDIV、 FSQRT指令推进到ED阶段后的FPU指令。（图8.43）

(3) 在FDIV、 FSQRT指令的末尾，发生运算的回写。这里再次使用E1阶段，因此正好在这以后的指令请求在E1阶段运行，以后的指令等待到FDIV、 FSQRT指令使用结束E1阶段为止。（图8.44）

(4) 只要先行的FDIV、 FSQRT使用ED阶段，紧接FDIV、 FSQRT之后的FDIV、 FSQRT就不进入ED阶段。

```
指令1(单精度)          IF   DF   E1   ED   ED   ED   ED   ED   ED   ED   ED   E1   E2   SF
(FDIV FR6,FR7)
指令2(单精度)               IF   DF   E1   E2   SF
(FADD FR8,FR10)
```

图 8.43 由 FDIV 引起的 E1 阶段竞争例

```
指令1(单精度)          IF   DF   E1   ED   ED   ED   ED   ED   ED   ED   ED   E1   E2   SF
(FDIV FR6,FR7)
其他的指令                                        :

指令2(单精度)                                          IF   DF   E1   E2   SF
(FADD FR8,FR10)
指令3(单精度)                                               IF   --   DF   E1   E2   SF
(FADD FR9,FR11)
```

图 8.44 由 FDIV 引起的 E1 阶段竞争例 2

在双精度算术运算指令中，对于作为源寄存器使用的寄存器，通过以前的指令进行写入并且以前的指令的等待时间小于等于 2 个周期时，这些指令的等待时间为 2 个。（图 8.45）

```
浮点加载存储指令(双精度)          IF   DF   EX   EX   NA   SF
(FMOV DR0, DR2)(等待时间1→等待时间2)
下一条浮点算术运算指令(双精度)  IF   --   --   DF   E1   E1   …   E1   E2   SF
(FADD  DR2,DR4)
```

图 8.45 紧接双精度算术运算之前的 1 个等待时间的指令例

后续的指令将双精度算术运算指令的目标寄存器作为源寄存器使用时，如果 FRn 的 n 是奇数就减少 1 个周期等待时间。（图 8.46）但是，如果 FRn 的 n 是偶数，等待时间就不减少。（图 8.47）

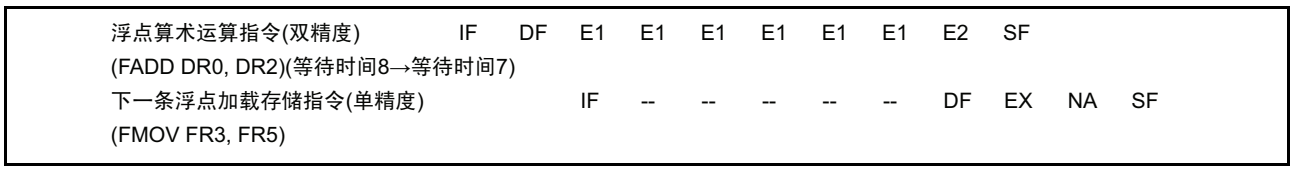


图 8.46 通过双精度算术运算指令减少等待时间的例子

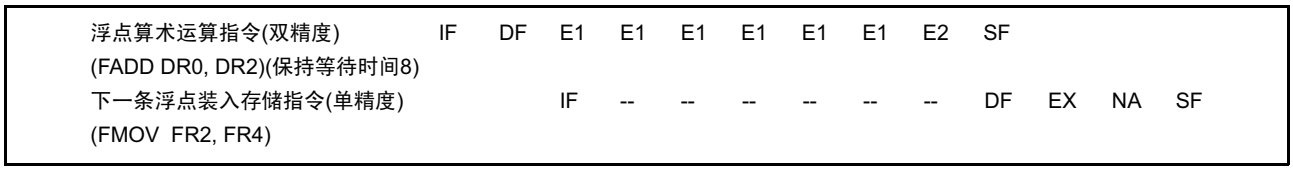


图 8.47 通过双精度算术运算指令不减少等待时间的例子

通过后续浮点数算术运算指令或者浮点加载存储指令写保存浮点数算术运算指令的结果的寄存器（FR0 ～ FR15、 FPUL）（作为目标寄存器使用）时，后续指令执行前保持等待。等待的周期数是之前的运算为 FDIV、 FSQRT 时的等待时间数 –1 的周期，除此以外的情况为等待时间数 –2 的周期。（图 8.48）（图 8.49）

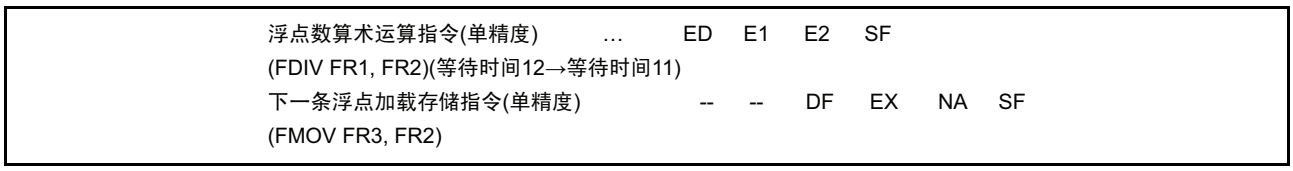


图 8.48 由重写引起的竞争例 （FDIV、 FSQRT）

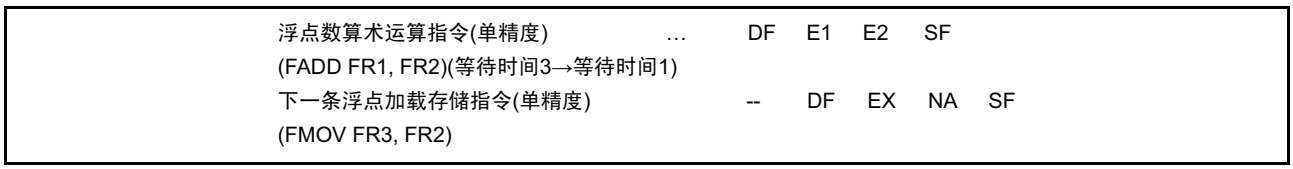


图 8.49 由重写引起的竞争例 （FDIV、 FSQRT 以外）

双精度 FADD、 FSUB、 FMUL 中对于作为源使用的寄存器，欲通过后续的指令写入时，后续的指令需要等待 2 个周期。（图 8.50）

图 8.50 紧接双精度运算之后，写入双精度指令源例

8.7 由乘法器引起的竞争

乘法指令、乘加指令、以及这些目标寄存器 （MACH、 MACL）的操作指令，使用乘法器。另外， STS FPUL,Rn、 STS FPSCR,Rn 指令使用乘法器的结果寄存器读取总线。在此，按如下区分指令，说明与各流水线竞争的详细内容。紧接指令之后的 （A/B/C）表示 （执行槽数 / 等待时间 / 锁槽数）。

○乘加指令

MAC.L （4/6/5）	IF	ID	EX	MA	MA	mm	mm	mm
MAC.W （3/5/4）	IF	ID	EX	MA	MA	mm	mm	

○乘法指令 （I）

DMUL.S、 DMUL.U、 MUL.L （2/3/2）	IF	ID	mm	mm	mm		
MULS.W、 MULU.W （1/2/1）	IF	ID	mm	mm			

○乘法指令 （II）（寄存器返回）

MULR （2/4/2）	IF	ID	mm	mm	mm	WB

○寄存器写入指令 （I）

CLRMAC、 LDS （1/2/1）	IF	ID	mm	mm	

○寄存器写入指令 （II）

LDS.L （1/3/2）	IF	ID	EX	MA	WB

○寄存器读取指令 （也含 STS FPUL,Rn、 STS FPSCR,Rn）

STS （1/2/0）	IF	ID	EX	WB	
STS.L （1/2/0）	IF	ID	EX	MA	

竞争的说明

多周期指令与普通指令同样会发生竞争。（图 8.51）详细内容请参照 “8.3.4 由多周期指令引起的竞争的详细内容”。

MAC.L @R1+, @R2+	IF	ID	EX	MA	MA	mm	mm	mm			
MAC.L @R3+, @R4+	IF	--	--	--	ID	EX	MA	MA	mm	mm	mm

【注】 MAC.L为执行速率4的指令。

图 8.51 使用乘法器的多周期指令例

使用乘法器的指令有以下规则。

(1) 将乘法结果作为源使用的指令的执行等待该指令的等待时间。（图8.52）但是，后续指令是MACH、MACL读取指令时，等待时间数–1的周期。（图8.53）另外，后续指令是乘加指令时不等待。（图8.54）

MULR R0,R4	IF	ID	mm	mm	mm	WB	
ADD R4,R	IF	--	--	--	--	ID	EX WB

图 8.52 紧接乘法之后参考结果寄存器例 1

MUL.L R2, R3	IF	ID	mm	mm	mm	
STS MACH, R4	IF	--	--	ID	EX	WB

图 8.53 紧接乘法之后参考结果寄存器例 2

MAC.W @R1+, @R2+	IF	ID	EX	MA	MA	mm	mm	
MAC.W @R3+, @R4+	IF	--	--	ID	EX	MA	MA	mm mm

图 8.54 紧接乘法之后参考结果寄存器例 3

(2) 当之前的指令锁住乘法器时，使用乘法器的指令之后的指令等到解锁后执行。（图8.55）

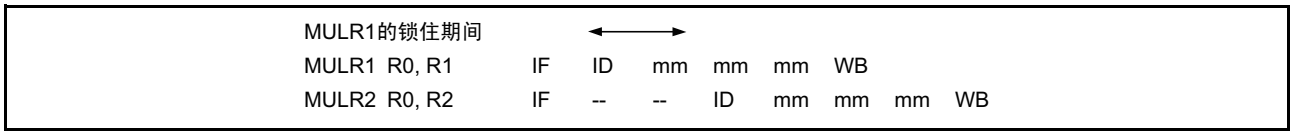


图 8.55 乘法器锁住竞争的例子

但是，后续指令是乘加指令时不需等待锁解除，只需等待与通常的多周期指令相同的状态期间后执行。（图8.56）

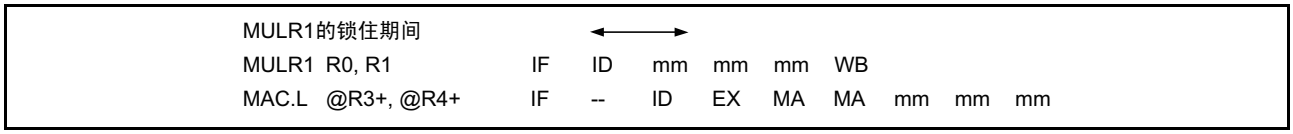


图 8.56 后续指令是乘加指令，不发生乘法器锁竞争的例子

另外，后续指令是寄存器写指令 （II）时，如果锁住期间还剩1个槽就执行该指令。（图8.57）

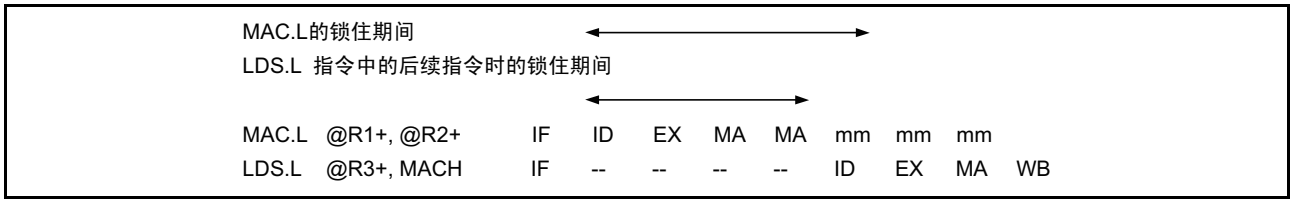


图 8.57 锁住期间时间还剩 1 时执行 LDS.L 指令例

另外， STS， STS.L指令不锁住乘法器。因此， STS指令和MUL.L指令等可并行执行。（图8.58）

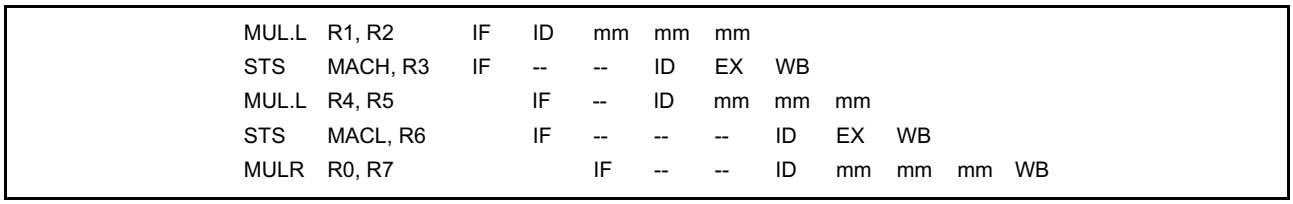


图 8.58 STS 指令和 MUL.L 指令并行执行例

(3) MULR指令与对于MACH、 MACL、 FPUL、 FPSCR的STS指令以及对于MACH、 MACL的STS.L指令共享结果寄存器读取总线，引起资源竞争。（MA、 WB阶段）因此，这些STS、 STS.L指令不能并行执行。（图8.59）
另外，紧接MULR指令之后配置STS、 STS.L指令时，同样引起WB阶段的竞争， STS、 STS.L指令等待3个周期后执行。（图8.60）

图 8.59 在 STS、 STS.L 情况下的竞争例

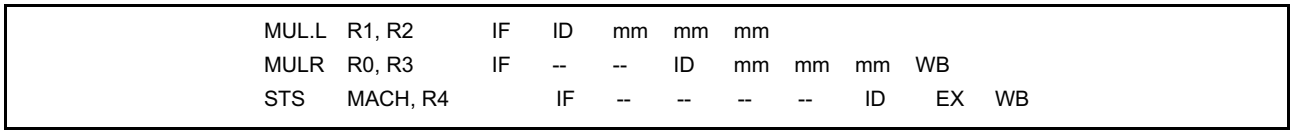


图 8.60 MULR 和 STS 的竞争例

8.8 编程的准则

为了提高指令的执行速度，编程时必须按如下进行编程：

(1) 将转移目标地址配置在存储器的长字边界。这样，可使紧接转移之后的并行执行高效进行。

(2) 将不使用与加载指令的目标寄存器相同的寄存器的指令配置在紧接着存储器加载指令之后的1～3条指令中。尽可能将使用目标寄存器的指令配置在第4条指令以后。

(3) 将不使用与结果寄存器相同的寄存器的指令配置在紧接32位乘法指令之后的1～3条指令中。

(4) 使紧接浮点数算术运算指令之后的1～（该指令的等待时间×2）条为止的指令不使用浮点数算术运算指令的目标寄存器。

8.9 各指令的流水线运行

以下，对各指令的流水线的运行进行说明。通过附加上述规则和并行执行可能性，可计算出程序的流水线流程和指令的执行状态数。

下述流水线图中，“指令 A”是要进行说明的指令。

发行指令的说明中，特别说明在考虑资源竞争时应如何处理该指令。

在并行执行的说明中，特别说明考虑并行执行性时应如何处理该指令。此处的记述是无寄存器竞争的情况。

指令的阶段数与执行状态数用以下格式表示。并且，在表中表示该指令不依存寄存器执行时的状态数。

指令的阶段数和执行状态数的格式

分类	区分	阶段数	执行状态	等待时间	竞争	指令
按照功能分类。	按照不同的运行区分指令。	指令的阶段数。	未发生竞争时的执行状态数。	到执行结果确定为止的执行状态数。	表示发生资源竞争。	用助记符表示对应的指令。

表 8.1 指令的阶段数与执行状态数

分类	区分	阶段数	执行状态	等待时间	竞争	指令	
数据传送指令	寄存器－寄存器间的传送指令	3	1	1	—	MOV	#imm,Rn
			1	0		MOV	Rm,Rn
			1	1		MOVA	@(disp,PC),R0
						MOVT	Rn
						MOVRT	Rn
						NOTT	
						SWAP.B	Rm,Rn
						SWAP.W	Rm,Rn
						XTRCT	Rm,Rn
					• 该指令是 32 位指令。	MOVI20	#imm,Rn
						MOVI20S	#imm20,Rn
	存储器加载指令	5	1	2	• 该指令使用存储器存取流水线。	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

分类	区分	阶段数	执行状态	等待时间	竞争	指令	
数据传送指令	存储器加载指令	5	1	2	• 该指令使用存储器存取流水线。	MOV.B	@-Rm,R0
						MOV.W	@-Rm,R0
						MOV.L	@-Rm,R0
						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
		5 ～ 20	1 ～ 16	2 ～ 17		MOVML.L	@R15+,Rn
						MOVMU.L	@R15+,Rn
		5	1	2	• 该指令是 32 位指令。 • 该指令使用存储器存取流水线。	MOV.B	@(disp12,Rm),Rn
						MOV.W	@(disp12,Rm),Rn
						MOV.L	@(disp12,Rm),Rn
						MOVU.B	@(disp12,Rm),Rn
						MOVU.W	@(disp12,Rm),Rn
	存储器存储指令	4	1	0	• 该指令使用存储器存取流水线。	MOV.B	Rm,@Rn
						MOV.W	Rm,@Rn
						MOV.L	Rm,@Rn
				1		MOV.B	Rm,@-Rn
						MOV.W	Rm,@-Rn
						MOV.L	Rm,@-Rn
						MOV.B	R0,@Rn+
						MOV.W	R0,@Rn+
						MOV.L	R0,@Rn+
				0		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)
		4 ～ 19	1 ～ 16	1 ～ 16		MOVML.L	Rm,@-R15
						MOVMU.L	Rm,@-R15

分类	区分	阶段数	执行状态	等待时间	竞争	指令	
数据传送指令	存储器存储指令	4	1	0	• 该指令是 32 位指令。 • 该指令使用存储器存取流水线。	MOV.B	Rm,@(disp12,Rn)
						MOV.W	Rm,@(disp12,Rn)
						MOV.L	Rm,@(disp12,Rn)
	PREF 指令	4	1	0	• 该指令使用存储器存取流水线。	PREF	@Rm
算术运算指令	寄存器间算术运算指令 (乘法指令除外)	3	1	1	—	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
	寄存器间算术运算指令 (乘法指令 DIVU、 DIVS 指令除外)	3	1	1	—	NEGC	Rm,Rn
						SUB	Rm,Rn
						SUBC	Rm,Rn
						SUBV	Rm,Rn
	CLIP 指令	3	1	1	—	CLIPU.B	Rn
						CLIPU.W	Rn
						CLIPS.B	Rn
						CLIPS.W	Rn
	乘加指令	7	3	4	• 该指令锁住乘法器 4 个槽的期间。	MAC.W	@Rm+,@Rn+

分类	区分	阶段数	执行状态	等待时间	竞争	指令
算术运算指令	双精度乘加指令	8	4	5	• 该指令锁住乘法器 5 个槽的期间。	MAC.L @Rm+,@Rn+
	乘法指令	4	1	2	• 该指令锁住乘法器 1 个槽的期间。	MULS.W Rm,Rn
						MULU.W Rm,Rn
	双精度乘法指令	5	2	3	• 该指令锁住乘法器 2 个槽的期间。	DMULS.L Rm,Rn
						DMULU.L Rm,Rn
						MUL.L Rm,Rn
		6	2	4		MULR R0,Rn
	DIVU、DIVS 指令	36	34	34	• 该指令使用移位流水线	DIVU R0,Rn
		38	36	36	※ DIVS 请写 “—”。	DIVS R0,Rn
逻辑运算指令	寄存器－寄存器间逻辑运算指令	3	1	1	—	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
	存储器逻辑运算指令	6	3	2	• 该指令使用存储器存取流水线。	AND.B #imm,@(R0,GBR)
						OR.B #imm,@(R0,GBR)
		5		3		TST.B #imm,@(R0,GBR)
		6		2		XOR.B #imm,@(R0,GBR)
	TAS 指令	6	3	3	• 该指令使用存储器存取流水线。	TAS.B @Rn
位操作指令	寄存器－寄存器间位运算指令	3	1	1	—	BLD #imm3,Rn
						BSET #imm3,Rn
						BCLR #imm3,Rn
						BST #imm3,Rn
	存储器－T 位间位运算指令	5	3	3	• 该指令是 32 位指令。 • 该指令使用存储器存取流水线。	BAND.B #imm3,@(disp12,Rn)
						BANDNOT.B #imm3,@(disp12,Rn)
						BOR.B #imm3,@(disp12,Rn)
						BORNOT.B #imm3,@(disp12,Rn)

分类	区分	阶段数	执行状态	等待时间	竞争	指令
位操作指令	存储器－T 位间位运算指令	5	3	3	• 该指令是 32 位指令。 • 该指令使用存储器存取流水线。	BLD.B #imm3,@(disp12,Rn)
						BLDNOT.B #imm3,@(disp12,Rn)
						BXOR.B #imm3,@(disp12,Rn)
	存储器位操作指令	6	3	2		BST.B #imm3,@(disp12,Rn)
						BCLR.B #imm3,@(disp12,Rn)
						BSET.B #imm3,@(disp12,Rn)
移位指令	移位指令	3	1	1	• 该指令使用移位流水线。	ROTL Rn
						ROTR Rn
						ROTCL Rn
						ROTCR Rn
						SHAL Rn
						SHAR Rn
						SHLL Rn
						SHLR Rn
						SHLL2 Rn
						SHLR2 Rn
						SHLL8 Rn
						SHLR8 Rn
						SHLL16 Rn
						SHLR16 Rn
						SHAD Rm,Rn
						SHLD Rm,Rn
转移指令	条件转移指令	3	3/1*1	3/1*1	• 该指令使用转移流水线。	BF label
						BT label
	带延迟条件转移指令	3	2/1*1	2/1*1	• 该指令使用转移流水线。	BF/S label
						BT/S label
	无条件转移指令	3	2	2	• 该指令使用转移流水线。	BRA label
						BRAF Rm
						BSR label
						BSRF Rm
						JMP @Rm
						JSR @Rm
						RTS
	不带无条件延迟的转移指令	3	3	3	• 该指令使用转移流水线。	JSR/N @Rm
						RTS/N
						RTV/N Rm

分类	区分	阶段数	执行状态	等待时间	竞争	指令
转移指令	不带无条件延迟的转移指令	5	5	5	• 该指令使用转移流水线。 • 该指令使用存储器存取流水线。	JSR/N@@(disp,TBR）
系统控制指令	系统控制 ALU 指令	3	1	1	—	CLRT
		5	3	2		LDC Rm,SR
		3	1	1	—	LDC Rm,GBR
						LDC Rm,TBR
						LDC Rm,VBR
						LDS Rm,PR
				0		NOP
						SETT
		4	2	2		STC SR,Rn
		3	1	1		STC GBR,Rn
						STC TBR,Rn
						STC VBR,Rn
						STS PR,Rn
	LDC.L 指令	7	5	4	• 该指令使用存储器存取流水线。	LDC.L @Rm+,SR
		5	1	2		LDC.L @Rm+,GBR
						LDC.L @Rm+,VBR
	STC.L 指令	5	2	2	• 该指令使用存储器存取流水线。	STC.L SR,@-Rn
		4	1	1		STC.L GBR,@-Rn
						STC.L VBR,@-Rn
	LDS.L 指令 (PR）	5	1	2	• 该指令使用存储器存取流水线。	LDS.L @Rm+,PR
	STS.L 指令 (PR）	4	1	1		STS.L PR,@-Rn
	寄存器 → MAC 传送指令	4	1	1	• 该指令锁住乘法器 1 个槽的期间。	CLRMAC
						LDS Rm,MACH
						LDS Rm,MACL
	存储器 → MAC 传送指令	5	1	2	• 该指令锁住乘法器 2 个槽的期间。	LDS.L @Rm+,MACH
						LDS.L @Rm+,MACL
	MAC→ 寄存器传送指令	4	1	2	• 该指令使用乘法结果的读取总线。	STS MACH,Rn
						STS MACL,Rn
	MAC→ 存储器传送指令	4	1	1	• 该指令使用乘法结果的读取总线。	STS.L MACH,@-Rn
						STS.L MACL,@-Rn
	RTE 指令	8	6	5	—	RTE
	RESBANK 指令	11/23*2	9/19*2	8/20*2	• BO 位为 1 时，使用存储器存取流水线。	RESBANK

分类	区分	阶段数	执行状态	等待时间	竞争	指令
系统控制指令	LDBANK 指令	8	6	5	—	LDBANK @Rm,R0
	STBANK 指令	9	7	6	—	STBANK R0,@Rn
	TRAP 指令	8	5	6	—	TRAPA #imm
	SLEEP 指令	7	5	0	—	SLEEP
FPU 加载存储指令	FPUL 加载指令	5	1	1	—	LDS Rm,FPUL
				2	• 该指令使用存储器存取流水线。	LDS.L @Rm+,FPUL
	FPSCR 加载指令	5	1	3	—	LDS Rm,FPSCR
				3	• 该指令使用存储器存取流水线。	LDS.L @Rm+,FPSCR
	FPUL 存储指令 (STS)	4	1	2	• 该指令使用乘法结果的读取总线。	STS FPUL,Rn
	FPUL 存储指令 (STS.L)	4	1	1	• 该指令使用存储器存取流水线。	STS.L FPUL,@-Rn
	FPSCR 存储指令 (STS)	4	1	2	• 该指令使用乘法结果的读取总线。	STS FPSCR,Rn
	FPSCR 存储指令 (STS.L)	4	1	1	• 该指令使用存储器存取流水线。	STS.L FPSCR,@-Rn
单精度浮点指令	浮点寄存器－寄存器间传送指令	5	1	0	• 该指令使用 FPU 加载存储流水线。	FLDS FRm,FPUL
						FMOV FRm,FRn
						FSTS FPUL,FRn
	浮点寄存器－立即指令	5	1	0	• 该指令使用 FPU 加载存储流水线。	FLDI0 FRn FLDI1 FRn
	FSCHG 指令	5	1	1	• 该指令使用 FPU 算术运算流水线。	FSCHG
	浮点寄存器加载指令	5	1	0/2*3	• 该指令使用 FPU 加载存储流水线与存储器存取流水线。	FMOV.S @Rm,FRn
			1	1/2*3		FMOV.S @Rm+,FRn
				0/2*3		FMOV.S @(R0,Rm),FRn

分类	区分	阶段数	执行状态	等待时间	竞争	指令	
单精度浮点指令	浮点寄存器存储指令	4	1	0/2*3	• 该指令是 32 位指令。 • 该指令使用 FPU 加载存储流水线与存储器存取流水线。	FMOV.S	@(disp12,Rm),FRn
				0	• 该指令使用 FPU 加载存储流水线与存储器存取流水线。	FMOV.S	FRm,@Rn
				1/0*3		FMOV.S	FRm,@-Rn
				0		FMOV.S	FRm,@ (R0,Rn)
					• 该指令是 32 位指令。 • 该指令使用 FPU 加载存储流水线与存储器存取流水线。	FMOV.S	FRm,@(disp12,Rn)
	浮点运算指令 (FDIV 除外)	5	1	3	• 该指令使用 FPU 算术运算流水线。	FADD	FRm,FRn
						FLOAT	FPUL,FRn
						FMAC	FR0,FRm,FRn
						FMUL	FRm,FRn
						FSUB	FRm,FR
						FTRC	FRm,FPUL
		5	1	0	• 该指令使用 FPU 加载存储流水线。	FABS	FRn
						FNEG	FRn
	浮点运算指令 (FDIV、FSQRT)	14 13	1 1	12 11	• 该指令使用 FPU 算术运算流水线、FPU 除法平方根运算流水线。	FDIV	FRm,FRn
						FSQRT	FRn
	浮点比较指令	4	1	2	• 该指令使用 FPU 算术运算流水线。	FCMP/EQ	FRm,FRn
						FCMP/GT	FRm,FRn
双精度浮点指令	浮点寄存器－寄存器间传送指令	6	2	1	• 该指令使用 FPU 加载存储流水线。	FMOV	DRm,DRn
	FCNVDS、FCNVSD 指令	5	1	4	• 该指令使用 FPU 算术运算流水线。	FCNVSD FCNVDS	FPUL,DRn DRm,FPUL

分类	区分	阶段数	执行状态	等待时间	竞争	指令	
双精度浮点指令	浮点寄存器加载指令	6	2	0/2/3/4*4	• 该指令使用 FPU 加载存储流水线与存储器存取流水线。	FMOV.D	@Rm,DRn
				1/2/3/4*4		FMOV.D	@Rm+,DRn
				0/2/3/4*4		FMOV.D	@(R0,Rm),DRn
				0/2/3/4	• 该指令是 32 位指令。 • 该指令使用 FPU 加载存储流水线与存储器存取流水线。	FMOV.D	@(disp12,Rm),DRn
	浮点寄存器存储指令	5	2	0	• 该指令使用 FPU 加载存储流水线与存储器存取流水线。	FMOV.D	DRm,@Rn
				1/0*3		FMOV.D	DRm,@-Rn
				0		FMOV.D	DRm,@ (R0,Rn)
					• 该指令是 32 位指令。 • 该指令使用 FPU 加载存储流水线与存储器存取流水线。	FMOV.D	DRm,@(disp12,Rn)
	浮点运算指令 (FDIV 除外)	10	1	0/8/7/8*4	• 该指令使用 FPU 算术运算流水线。	FADD	DRm,DRn
						FMUL	DRm,DRn
						FSUB	DRm,DRn
		6	1	0/4*3		FTRC	DRm,FPUL
		6	1	0/4/3/4*4		FLOAT	FPUL,DRn
		5	1	0	• 该指令使用 FPU 加载存储流水线。	FABS	DRn
						FNEG	DRn
	浮点运算指令 (FDIV、FSQRT)	27	1	0/25/24/25*4	• 该指令使用 FPU 算术运算流水线、 FPU 除法平方根运算流水线。	FDIV	DRm,DRn
		26	1	0/24/23/24*4		FSQRT	DRn
	浮点比较指令	5	2	3	• 该指令使用 FPU 算术运算流水线。	FCMP/EQ	DRm,DRn
						FCMP/GT	DRm,DRn

【注】 *1 不转移时为 1 个状态。

*2 按照以下顺序表示阶段数、执行状态、等待时间：

（1） BO位为0时

（2） BO位为1时

*3 按照以下顺序表示等待时间。

（1） CPU寄存器

（2） FPU寄存器

*4 按照以下的顺序表示等待时间：

（1） CPU寄存器

（2）作为单精度寄存器使用时的FRn的偶数

（3）作为单精度寄存器使用时的FRn的奇数

（4）作为双精度寄存器使用时

8.9.1 数据传送指令

(1) 寄存器－寄存器间的传送指令 （MOV Rm, Rn）

- 指令种类

```
MOV     Rm,Rn
```

- 流水线

	←→	←→	←→	←→	←→	槽
指令 A	IF	ID	EX			
下一条指令	IF	ID	EX	…		
下下一条指令		IF	ID	EX	…	

- 运行说明

流水线有 IF、 ID 和 EX 的 3 个阶段。 EX 阶段中，经过 ALU 传送数据。

- 指令发行

该指令不引起资源竞争。

- 并行执行可能性

该指令是等待时间 0 的指令。该指令作为先行指令执行并且后续指令使用 Rn 时也可并行执行。

(2) 寄存器－寄存器间传送指令 （20 位立即值）

- 指令种类

```
MOVI20      #imm20,Rn
MOVI20S     #imm20,Rn
```

- 流水线

	←→	←→	←→	←→	←→	槽
指令 A	IF	ID	EX			
下一条指令		IF	ID	EX	…	
下下一条指令		IF	ID	EX	…	

- 运行说明

流水线有 IF、 ID 和 EX 的 3 个阶段。 EX 阶段中，经过 ALU 传送数据。

- 指令发行

该指令不引起资源竞争。

- 并行执行可能性

该指令是 32 位指令，不可并行执行。（参照 “8.3.5 由 32 位指令引起的竞争的详细内容”）。

(3) 寄存器－寄存器间传送指令 （MOV Rm,Rn、 MOVI20、 MOVI20S 除外）

• 指令种类

```
MOV         #imm,Rn
MOVA        @(disp,PC),R0
MOVT        Rn
MOVRT       Rn
SWAP.B      Rm,Rn
SWAP.W      Rm,Rn
XTRCT       Rm,Rn
NOTT
```

• 流水线

	←→	←→	←→	←→	←→	槽
指令 A	IF	ID	EX			
下一条指令	IF	ID	EX	…		
下下一条指令		IF	ID	EX	…	

• 运行说明

流水线有 IF、 ID 和 EX 的 3 个阶段。 EX 阶段中，经过 ALU 传送数据。

• 指令发行

SWAP.B、 SWAP.W、 XTRCT 指令使用移位器。

其他指令不引起资源竞争。

• 并行执行可能性

无特别记载事项。

(4) 存储器加载指令

• 指令种类

```
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       @-Rm,R0
MOV.W       @-Rm,R0
MOV.L       @-Rm,R0
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
```

• 流水线

	←→	←→	←→	←→	←→	槽
指令 A	IF	ID	EX	MA	WB	
下一条指令	IF	ID	EX	…		
下下一条指令		IF	ID	EX	…	

• 运行说明

流水线有 IF、ID、EX、MA、WB 的 5 个阶段。如果将使用该指令的目标寄存器的指令设置在该指令的 1 ～ 3 条指令之后，有时会发生竞争 （参照 **“8.5 存储器加载指令对流水线的影响”**）。

• 指令发行

该指令使用存储器存取流水线。

• 并行执行可能性

无特别记载事项。

(5) 存储器加载指令 （12 位位移量）

• 指令种类

```
MOV.B       @(disp12,Rm),Rn
MOV.W       @(disp12,Rm),Rn
MOV.L       @(disp12,Rm),Rn
MOVU.B      @(disp12,Rm),Rn
MOVU.W      @(disp12,Rm),Rn
```

• 流水线

	←→	←→	←→	←→	←→	槽
指令 A	IF	ID	EX	MA	WB	
下一条指令		IF	ID	EX	…	
下下一条指令		IF	ID	EX	…	

• 运行说明

流水线有 IF、ID、EX、MA、WB 的 5 个阶段。如果将使用该指令的目标寄存器的指令设置在该指令的 1 ～ 2 条指令之后，有时会发生竞争 （参照 **“8.5 存储器加载指令对流水线的影响”**）。

• 指令发行

该指令使用存储器存取流水线。

• 并行执行可能性

该指令为 32 位指令，不可并行执行 （参照 **“8.3.5 由 32 位指令引起的竞争的详细内容”**）。

(6) 存储器加载指令 （MOVMU.L、 MOVML.L）

• 指令种类

```
MOVMU.L     @R15+,Rn
MOVML.L     @R15+,Rn
```

• 流水线

	←→	←→	←→	←→	←→	←→	←→	←→	←→	槽
指令 A	IF	ID	EX	MA	…	MA	MA	MA	WB	
下一条指令	IF	--	--	--	…	ID	EX	…		
下下一条指令		IF	--	--	…	--	ID	EX	…	

• 运行说明

该指令是执行从堆栈返回的指令。流水线有 IF、 ID、 EX、 MA、 MA、 MA、… MA、 WB 阶段，根据需要重复 MA。如果将使用该指令的目标寄存器的指令设置在该指令的 1 ～ 3 条指令之后，有时会发生竞争 （参照 “8.5 存储器加载指令对流水线的影响”）。

• 指令发行

当该指令被解码时，如果 CPU 流水线内有未完成的指令，该指令的执行就会延迟。

该指令使用存储器存取流水线。

• 并行执行可能性

该指令是多周期指令，不可与后续指令并行执行 （参照 “8.3.4 由多周期指令引起的竞争的详细内容”）。

(7) 存储器存储指令

• 指令种类

```
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,@Rn+
MOV.W   R0,@Rn+
MOV.L   R0,@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)
```

• 流水线

	←→	←→	←→	←→	←→	槽
指令 A	IF	ID	EX	MA		
下一条指令	IF	ID	EX	…		
下下一条指令		IF	ID	EX	…	

• 运行说明

流水线有 IF、 ID、 EX 和 MA 的 4 个阶段。因为不需要将数据返回寄存器，所以无 WB 阶段。

• 指令发行

该指令使用存储器存取流水线。

• 并行执行可能性

无特别记载事项。

(8) 存储器存储指令 （12 位位移量）

• 指令种类

```
MOV.B    Rm,@(disp12,Rn)
MOV.W    Rm,@(disp12,Rn)
MOV.L    Rm,@(disp12,Rn)
```

• 流水线

	←→	←→	←→	←→	←→	槽
指令 A	IF	ID	EX	MA		
下一条指令		IF	ID	EX	…	
下下一条指令		IF	ID	EX	…	

• 运行说明

流水线有 IF、 ID、 EX 和 MA 的 4 个阶段。因为不需要将数据返回寄存器，所以无 WB 阶段。

• 指令发行

该指令使用存储器存取流水线。

• 并行执行可能性

该指令为 32 位指令，不可并行执行 （参照 “8.3.5 由 32 位指令引起的竞争的详细内容”）。

(9) 存储器存储指令 （MOVMU.L、 MOVML.L）

• 指令种类

```
MOVMU.L    Rm,@-R15
MOVML.L    Rm,@-R15
```

• 流水线

	←→	←→	←→	←→	←→	←→	←→	←→	槽
指令 A	IF	ID	EX	MA	…	MA	MA	MA	
下一条指令	IF	--	--	--	…	ID	EX	…	
下下一条指令		IF	--	--	…	--	ID	EX	…

• 运行说明

该指令是执行向堆栈保存的指令。流水线有 IF、ID、EX、MA、MA、MA、… MA 阶段，根据需要重复 MA。因为不需要将数据返回寄存器，所以无 WB 阶段。

• 指令发行

当该指令被解码时，如果 CPU 流水线内有未完成的指令，该指令的执行就会延迟。

该指令使用存储器存取流水线。

• 并行执行可能性

该指令为多周期指令，不可与后续指令并行执行。

(10) PREF 指令

- 指令种类

  PREF @Rm

- 流水线

	←→	←→	←→	←→	←→	槽
指令 A	IF	ID	EX	MA		
下一条指令	IF	ID	EX	…		
下下一条指令		IF	ID	EX	…	

- 运行说明

  流水线有 IF、 ID、 EX 和 MA 的 4 个阶段。因为不需要将数据返回寄存器，所以无 WB 阶段。

- 指令发行

  该指令使用存储器存取流水线。

- 并行执行可能性

  无特别记载事项。

8.9.2 算术运算指令

(1) 寄存器间算术运算指令 （乘法、 DIVU、 DIVS 指令除外）

- 指令种类

```
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
CLIPU.B     Rn
CLIPU.W     Rn
CLIPS.B     Rn
CLIPS.W     Rn
```

- 流水线

	←→	←→	←→	←→	←→	槽
指令 A	IF	ID	EX			
下一条指令	IF	ID	EX	…		
下下一条指令		IF	ID	EX	…	

- 运行说明

流水线有 IF、 ID、 EX 的 3 个阶段。 EX 阶段中，经过 ALU 完成数据运算。

- 指令发行

EXTS.B、 EXTS.W、 EXTU.B、 EXTU.W 指令使用移位器。
其他指令不引起资源竞争。

- 并行执行可能性

在 CLIP 指令中，不发生 CS 位盖写竞争，可相互并行执行。

(2) 乘加指令

• 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	…			
下下一条指令		IF	--	--	ID	EX	…		

• 运行说明

流水线有 IF、 ID、 EX、 MA、 MA、 mm、 mm 的 7 个阶段。 mm 表示乘法器正在运行的状态。

一般流水线的详细内容，请参照 “8.7 由乘法器引起的竞争”。该指令有 3 个执行槽、 5 个等待时间、 4 个锁槽。关于该指令的流水线和乘法器的相关指令连续时的详细内容例如下所示：

(a) 在紧接着MAC.W指令之后连续出现MAC.W、 MAC.L指令时乘法器不竞争。

	←→	←→	←→	←→	←→	←→	←→	←→	槽
MAC.W @Rm+, @Rn+	IF	ID	EX	MA	MA	mm	mm		
MAC.W @Rm+, @Rn+	IF	--	--	ID	EX	MA	MA	mm	mm
下下一条指令		IF	--	--	--	ID	EX	…	

(b) 在紧接着MAC.W指令之后连续出现MULS.W、 MULU.W、 DMULS.L、 DMULU.L、 MUL.L、 MULR、STS （寄存器）、 STS.L （存储器）、 LDS （寄存器）指令时，由于MAC.W指令锁住乘法器，因此再停止2个槽的期间。

	←→	←→	←→	←→	←→	←→	←→	←→	槽
MAC.W @Rm+, @Rn+	IF	ID	EX	MA	MA	mm	mm		
STS MACL, Rn	IF	--	--	--	--	ID	EX	WB	
下下一条指令		IF	--	--	--	ID	EX	…	

(c) 在紧接着MAC.W指令之后出现LDS.L （存储器）指令时，延迟执行MAC指令的执行状态 （3个槽）的期间。

	←→	←→	←→	←→	←→	←→	←→	←→	槽
MAC.W @Rm+, @Rn+	IF	ID	EX	MA	MA	mm	mm		
LDS.L @Rn+,MACL	IF	--	--	--	ID	EX	MA	WB	
下下一条指令		IF	--	--	ID	EX	…		

• 指令发行

该指令使用存储器存取流水线。

该指令使用乘法器。

即使锁住乘法器仍执行该指令。

该指令锁住乘法器 4 个槽的期间。

• 并行执行可能性

该指令为多周期指令，不可与后续指令并行执行。（参照 “8.3.4 由多周期指令引起的竞争的详细内容”）

(3) 双精度乘加指令

• 指令种类

MAC.L @Rm+,@Rn+

• 流水线

	←→	←→	←→	←→	←→	←→	←→	←→	槽
指令 A	IF	ID	EX	MA	MA	mm	mm	mm	
下一条指令	IF	--	--	--	ID	EX	…		
下下一条指令		IF	--	--	--	ID	EX	…	

• 运行说明

流水线有 IF、 ID、 EX、 MA、 MA、 mm、 mm、 mm 的 8 个阶段。

mm 表示乘法器正在运行的状态。

一般流水线的详细内容，请参照 “8.7 由乘法器引起的竞争”。该指令有 4 个执行槽、 6 个等待时间、 5 个锁槽。关于该指令的流水线和乘法器的相关指令连续时的详细内容例如下所示：

(a) 在紧接着MAC.L指令之后连续出现MAC.L、 MAC.W指令时乘法器不竞争。

	←→	←→	←→	←→	←→	←→	←→	←→	槽		
MAC.L @Rm+, @Rn+	IF	ID	EX	MA	MA	mm	mm	mm			
MAC.L @Rm+, @Rn+	IF	--	--	--	ID	EX	MA	MA	mm	mm	mm
下下一条指令		IF	--	--	--	--	--	ID	EX	…	

(b) 在紧接着MAC.W指令之后连续出现MULS.W、 MULU.W、 DMULS.L、 DMULU.L、 MUL.L、 MULR、STS （寄存器）、 STS.L （存储器）、 LDS （寄存器）指令时，由于MAC.L指令锁住乘法器，因此再停止2个槽的期间。

	←→	←→	←→	←→	←→	←→	←→	←→	槽
MAC.L @Rm+, @Rn+	IF	ID	EX	MA	MA	mm	mm	mm	
STS MACH, Rn	IF	--	--	--	--	--	ID	EX	WB
下下一条指令		IF	--	--	--	--	ID	EX	…

(c) 在紧接着MAC.W指令之后出现LDS.L （存储器）指令时，延迟执行MAC指令的执行状态 （4个槽）的期间。

	←→	←→	←→	←→	←→	←→	←→	←→	槽
MAC.L @Rm+, @Rn+	IF	ID	EX	MA	MA	mm	mm	mm	
LDS.L @Rn+,MACL	IF	--	--	--	--	ID	EX	MA	WB
下下一条指令		IF	--	--	--	ID	EX	…	

• 指令发行

该指令使用存储器存取流水线。
该指令使用乘法器。
即使锁住乘法器仍执行该指令。
该指令锁住乘法器 5 个槽的期间。

• 并行执行可能性

该指令为多周期指令，不可与后续指令并行执行。（参照 “8.3.4 由多周期指令引起的竞争的详细内容”）。

(4) 乘法指令

• 指令种类

```
MULS.W    Rm,Rn
MULU.W    Rm,Rn
```

• 流水线

	←→	←→	←→	←→	←→	槽
指令 A	IF	ID	mm	mm		
下一条指令	IF	ID	EX	…		
下下一条指令		IF	ID	EX	…	

• 运行说明

流水线有 IF、 ID、 mm、 mm 的 4 个阶段。 mm 表示乘法器正在运行的状态。

一般流水线的详细内容，请参照 “8.7 由乘法器引起的竞争”。该指令有 1 个执行槽、 2 个等待时间、 1 个锁槽。关于该指令的流水线、乘法器相关的指令连续时的详细内容例如下所示：

(a) 在紧接着MULS.W指令之后连续出现MAC.W、 MAC.L指令时乘法器不竞争。

	←→	←→	←→	←→	←→	←→	←→	←→	槽
MULS.W Rm, Rn	IF	ID	mm	mm					
MAC.W @Rm+, @Rn+	IF	ID	EX	MA	MA	mm	mm		
下下一条指令		IF	--	ID	EX	…			

(b) 在紧接着MULS.W指令之后连续出现MULS.W、 MULU.W、 DMULS.L、 DMULU.L、 MUL.L、 MULR、STS （寄存器）、 STS.L （存储器）、 LDS （寄存器）指令时，由于MULS.W指令锁住乘法器，因此不可并行执行。

	←→	←→	←→	←→	←→	←→	槽
MULS.W Rm, Rn	IF	ID	mm	mm			
STS MACL,Rn	IF	--	ID	EX	WB		
下下一条指令		IF	ID	EX	…		

(c) 在紧接着MULS.W指令之后出现LDS.L （存储器）指令时， MULS.W指令锁住乘法器，因此不可并行执行

	←→	←→	←→	←→	←→	←→	槽
MULS.W Rm, Rn	IF	ID	mm	mm			
LDS.L @Rn+,MACL	IF	--	ID	EX	MA	WB	
下下一条指令		IF	ID	EX	…		

- 指令发行

  该指令使用乘法器。

  该指令锁住乘法器 1 个槽的期间。

- 并行执行可能性

  无特别记载事项。

(5) 双精度的乘法指令

- 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	mm	mm	mm	
下一条指令	IF	--	ID	EX	…	
下下一条指令		IF	--	ID	EX	…

- 运行说明

流水线有 IF、 ID、 mm、 mm、 mm 的 5 个阶段。 mm 表示乘法器正在运行的状态。

一般流水线的详细内容，请参照 “8.7 由乘法器引起的竞争”。该指令有 2 个执行槽、 3 个等待时间， 2 个锁槽。关于该指令的流水线以及乘法器相关的指令连续时的详细内容例如下所示：

(a) 在紧接着MUL.L的指令之后连续出现MAC.W、 MAC.L指令时乘法器不竞争。

	←→	←→	←→	←→	←→	←→	←→	←→	槽
MUL.L Rm, Rn	IF	ID	mm	mm	mm				
MAC.L @Rm+, @Rn+	IF	--	ID	EX	MA	MA	mm	mm	mm
下下一条指令		IF	--	--	--	ID	EX	…	

(b) 在紧接着MUL.L指令之后连续的出现MULS.W、 MULU.W、 DMULS.L、 DMULU.L、 MUL.L、 MULR、 STS （寄存器）、 STS.L （存储器）、 LDS （寄存器）指令时，由于MUL.L指令锁住乘法器，因此再停止2个槽的期间。

	←→	←→	←→	←→	←→	←→	←→	←→	槽
MUL.L Rm, Rn	IF	ID	mm	mm	mm				
STS MACL,Rn	IF	--	--	ID	EX	WB			
下下一条指令		IF	--	ID	EX	…			

(c) 在紧接着MUL.L指令之后出现LDS.L （存储器）指令时，延迟执行MUL.L指令的执行状态 （2个槽）的期间。

	←→	←→	←→	←→	←→	←→	←→	←→	槽
MUL.L Rm, Rn	IF	ID	mm	mm	mm				
LDS.L @Rn+,MACL	IF	--	--	ID	EX	MA	WB		
下下一条指令		IF	--	ID	EX	…			

- 指令发行

该指令使用乘法器。

该指令锁住乘法器 2 个槽的期间。

- 并行执行可能性

该指令为多周期指令，不可与后续指令并行执行。（参照 “8.3.4 由多周期指令引起的竞争的详细内容”）。

(6) 双精度乘法指令 （普通寄存器返回）

- 指令种类

```
MULR        R0,Rn
```

- 流水线

	←→	←→	←→	←→	←→	←→	←→	槽
指令 A	IF	ID	mm	mm	mm	WB		
下一条指令	IF	--	ID	EX	…			
下下一条指令		IF	ID	EX	…			

- 运行说明

流水线有 IF、 ID、 mm、 mm、 mm、 WB 的 6 个阶段。 mm 表示乘法器正在运行的状态。

一般流水线的详细内容，请参照 “8.7 由乘法器引起的竞争”。该指令有 2 个执行槽、 4 个等待时间、 1 个锁槽。关于该指令的流水线、乘法器相关的指令连续时的详细内容例如下所示：

(a) 在紧接着MULR指令之后连续出现MAC.W、 MAC.L指令时乘法器不竞争。

	←→	←→	←→	←→	←→	←→	←→	←→	槽
MULR R0, Rn	IF	ID	mm	mm	mm	WB			
MAC.L @Rm+, @Rn+	IF	--	ID	EX	MA	MA	mm	mm	mm
下下一条指令		IF	--	--	--	ID	EX	…	

(b) 在紧接着MULR指令之后连续出现MULS.W、MULU.W、DMULS.L、DMULU.L、MUL.L、MULR、STS （寄存器）、 STS.L （存储器）、 LDS （寄存器）指令时，由于MULR指令锁住乘法器，因此再停止1个槽的期间。

	←→	←→	←→	←→	←→	←→	←→	←→	槽
MULR R0, Rn	IF	ID	mm	mm	mm	WB			
MULR R0, Rn	IF	--	--	ID	mm	mm	mm	WB	
下下一条指令		IF	--	ID	EX	…			

(c) 在紧接着MULR指令之后连续出现STS （寄存器）、 STS.L （存储器）指令时，由于MULR指令锁住乘法器，并引起乘法结果读取总线的竞争，因此再停止2个槽期间。

	←→	←→	←→	←→	←→	←→	←→	←→	槽
MULR R0, Rn	IF	ID	mm	mm	mm	WB			
STS MACL,Rn	IF	--	--	--	ID	EX	WB		
下下一条指令		IF	--	--	ID	EX	…		

(d) 在紧接着MULR指令之后出现LDS.L （存储器）指令时，延迟执行MULR指令的执行状态 （2个槽）的期间。

	←→	←→	←→	←→	←→	←→	←→	←→	槽
MULR R0, Rn	IF	ID	mm	mm	mm	WB			
LDS.L @Rn+,MACL	IF	--	--	ID	EX	MA	WB		
下下一条指令		IF	--	ID	EX	…			

- 指令发行

该指令使用乘法器。

该指令锁住乘法器 2 个槽的期间。

该指令使用乘法结果的读取总线。

- 并行执行可能性

该指令为多周期指令，不可与后续指令并行执行。（参照 “8.3.4 由多周期指令引起的竞争的详细内容”）。

(7) DIVU 指令

• 指令种类

```
DIVU        R0,Rn
```

• 流水线

	←→	←→	←→	←→	←→	←→	←→	槽
指令 A	IF	ID	EX	…	EX	EX		
下一条指令	IF	--	--	--	ID	EX	…	
下下一条指令		IF	--	--	--	ID	EX	…

• 运行说明

流水线有 IF、ID、EX、EX、EX、EX、EX、EX、EX、EX、EX、EX、EX、EX、EX、EX、EX、EX、EX、EX、EX、EX、EX、EX、EX、EX、EX、EX、EX、EX、EX、EX、EX、EX、EX、EX、EX、EX、EX、EX、EX、EX 的 36 个阶段。EX 阶段中，经过 ALU 完成数据运算。

• 指令发行

该指令使用移位流水线。

• 并行执行可能性

该指令为多周期指令，不可与后续指令并行执行。

(8) DIVS 指令

• 指令种类

```
DIVS        R0,Rn
```

• 流水线

	←→	←→	←→	←→	←→	←→	←→	槽
指令 A	IF	ID	EX	…	EX	EX		
下一条指令	IF	--	--	--	ID	EX	…	
下下一条指令		IF	--	--	--	ID	EX	…

• 运行说明

流水线有 IF、ID、EX、EX、EX、EX、EX、EX、EX、EX、EX、EX、EX、EX、EX、EX、EX、EX、EX、EX、EX、EX、EX、EX、EX、EX、EX、EX、EX、EX、EX、EX、EX、EX、EX、EX、EX、EX、EX、EX、EX、EX、EX、EX 的 38 个阶段。 EX 阶段中，经过 ALU 完成数据运算。

• 指令发行

该指令不引起资源竞争。

• 并行执行可能性

该指令为多周期指令，不可与后续指令并行执行。

8.9.3 逻辑运算指令

(1) 寄存器－寄存器间逻辑运算指令

- 指令种类

```
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
```

- 流水线

	←→	←→	←→	←→	←→	槽
指令 A	IF	ID	EX			
下一条指令	IF	ID	EX	…		
下下一条指令		IF	ID	EX	…	

- 运行说明

流水线有 IF、 ID、 EX 的 3 个阶段。 EX 阶段中，经过 ALU 完成数据运算。

- 指令发行

该指令不引起资源竞争。

- 并行执行可能性

无特别记载事项。

(2) 存储器逻辑运算指令

- 指令种类

```
AND.B   #imm,@(R0,GBR)
OR.B    #imm,@(R0,GBR)
XOR.B   #imm,@(R0,GBR)
```

- 流水线

	←→	←→	←→	←→	←→	←→	槽
指令 A	IF	ID	EX	MA	EX	MA	
下一条指令	IF	--	--	ID	EX	…	
下下一条指令		IF	--	--	ID	EX	…

- 运行说明

流水线有 IF、 ID、 EX、 MA、 EX、 MA 的 6 个阶段。

- 指令发行

该指令使用存储器存取流水线。

- 并行执行可能性

该指令为多周期指令，不可与后续指令并行执行。（参照 “8.3.4 由多周期指令引起的竞争的详细内容”）。

(3) 存储器逻辑运算指令

• 指令种类

TST.B #imm,@(R0,GBR)

• 流水线

	←→	←→	←→	←→	←→	槽	
指令 A	IF	ID	EX	MA	EX		
下一条指令	IF	--	--	ID	EX	…	
下下一条指令		IF	--	--	ID	EX	…

• 运行说明

流水线有 IF、 ID、 EX、 MA、 EX 的 5 个阶段。

• 指令发行

该指令使用存储器存取流水线。

• 并行执行可能性

该指令为多周期指令，不可与后续指令并行执行。（参照 “8.3.4 由多周期指令引起的竞争的详细内容”）。

(4) TAS 指令

• 指令种类

TAS.B @Rn

• 流水线

	←→	←→	←→	←→	←→	←→	槽
指令 A	IF	ID	EX	MA	EX	MA	
下一条指令	IF	--	--	ID	EX	…	
下下一条指令		IF	--	--	ID	EX	…

• 运行说明

流水线有 IIF、 ID、 EX、 MA、 EX、 MA 的 6 个阶段。

• 指令发行

该指令使用存储器存取流水线。

• 并行执行可能性

该指令为多周期指令，不可与后续指令并行执行。

(5) 寄存器－寄存器间位运算指令

- 指令种类

```
BLD       #imm3,Rn
BSET      #imm3,Rn
BCLR      #imm3,Rn
BST       #imm3,Rn
```

- 流水线

	←→	←→	←→	←→	←→	槽
指令 A	IF	ID	EX			
下一条指令	IF	ID	EX	…		
下下一条指令		IF	ID	EX	…	

- 运行说明

流水线有 IF、 ID、 EX 的 3 个阶段。 EX 阶段中，经过 ALU 完成数据运算。

- 指令发行

该指令不引起资源竞争。

- 并行执行可能性

无特别记载事项。

(6) 存储器－ T 位间位运算指令

- 指令种类

```
BAND.B    #imm3,@(disp12,Rn)
BANDNOT.B #imm3,@(disp12,Rn)
BLD.B     #imm3,@(disp12,Rn)
BLDNOT.B  #imm3,@(disp12,Rn)
BOR.B     #imm3,@(disp12,Rn)
BORNOT.B  #imm3,@(disp12,Rn)
BXOR.B    #imm3,@(disp12,Rn)
```

- 流水线

	←→	←→	←→	←→	←→	←→	槽
指令 A	IF	ID	EX	MA	EX		
下一条指令		IF	--	ID	EX	…	
下下一条指令		IF	--	--	ID	EX	…

- 运行说明

流水线有 IF、 ID、 EX、 MA、 EX 的 5 个阶段。

- 指令发行

该指令使用存储器存取流水线。

• 并行执行可能性

该指令为 32 位指令，不可并行执行。该指令的后续指令为 BAND.B、 BANDNOT.B、 BLD.B、 BLDNOT.B、 BOR.B、 BORNOT.B、 BXOR.B 指令中任意一条时，最终阶段可与后续指令并行执行。不是这些指令时，最终阶段与后续指令不可并行执行。（参照 “8.3.5 由 32 位指令引起的竞争的详细内容”）。

	←→	←→	←→	←→	←→	←→		槽
BAND.B #imm, @(disp12, Rn)	IF	ID	EX	MA	EX			
BOR.B #imm, @(disp12, Rn)		IF	--	ID	EX	…		
BANDNOT.B #imm, @(disp12, Rn)			IF	--	--	ID	EX	…

	←→	←→	←→	←→	←→	←→	槽
BAND.B #imm, @(disp12, Rn)	IF	ID	EX	MA	EX		
ADD Rm, Rn		IF	--	--	ID	EX	…
下下一条指令		IF	--	--	ID	EX	…

	←→	←→	←→	←→	←→	←→	槽
BAND.B #imm, @(disp12, Rn)	IF	ID	EX	MA	EX		
ROTCL		IF	--	--	ID	EX	
BAND.B #imm, @(disp12, Rn)			IF	--	--	ID	EX
下下一条指令			IF	--	--	--	--

(7) 存储器位操作指令

• 指令种类

```
BCLR.B      #imm3,@(disp12,Rn)
BSET.B      #imm3,@(disp12,Rn)
BST.B       #imm3,@(disp12,Rn)
```

• 流水线

	←→	←→	←→	←→	←→	←→	槽	
指令 A	IF	ID	EX	MA	EX	MA		
下一条指令		IF	--	--	ID	EX	…	
下下一条指令		IF	--	--	--	ID	EX	…

• 运行说明

流水线有 IF、 ID、 EX、 MA、 EX、 MA 的 6 个阶段。

• 指令发行

该指令使用存储器存取流水线。

• 并行执行可能性

该指令为 32 位指令，不可并行执行。（参照 “8.3.5 由 32 位指令引起的竞争的详细内容”）。

8.9.4 移位指令

- 指令种类

ROTL	Rn
ROTR	Rn
ROTCL	Rn
ROTCR	Rn
SHAL	Rn
SHAR	Rn
SHLL	Rn
SHLR	Rn
SHLL2	Rn
SHLR2	Rn
SHLL8	Rn
SHLR8	Rn
SHLL16	Rn
SHLR16	Rn
SHAD	Rm,Rn
SHLD	Rm,Rn

- 流水线

	←→	←→	←→	←→	←→	槽
指令 A	IF	ID	EX			
下一条指令	IF	ID	EX	…		
下下一条指令		IF	ID	EX	…	

- 运行说明

  流水线有 IF、 ID、 EX 的 3 个阶段。 EX 阶段中，经过移位器完成数据运算。

- 指令发行

  该指令使用移位流水线。

- 并行执行可能性

  无特别记载事项。

8.9.5 转移指令

(1) 条件转移指令

- 指令种类

```
BF          label
BT          label
```

- 流水线

(a) 当条件成立时

	←→	←→	←→	←→	←→	槽	
指令 A	IF	ID	EX				
下一条指令	IF		…		(取指令，但不执行)		
下下一条指令		IF	…		(取指令，但不执行)		
下下下一条指令		IF	…		(取指令，但不执行)		
转移目标指令			--	IF	ID	EX	…

(b) 当条件不成立时

	←→	←→	←→	←→	←→	槽
指令 A	IF	ID	EX			
下一条指令	IF	ID	EX			
下下一条指令		IF	ID	EX	…	
下下下一条指令		IF	ID	EX	…	

- 运行说明

流水线有 IF、 ID、 EX 的 3 个阶段。在 ID 阶段进行条件判断。条件转移指令不是延迟转移指令。

(a) 当条件成立时

EX 阶段中，计算转移目标指令。到此为无效的被取指令全部不执行。转移指令的从指令 A 的 EX 阶段所在槽的下一个槽开始。

(b) 当条件不成立时

如果在 ID 阶段判断出条件不成立，就在 EX 阶段不进行任何运行，而继续取下一条指令并执行。典型的流水线如下所示：

先行指令为 CMP 指令时，延迟执行 1 个周期。

	←→	←→	←→	←→	←→	槽
CMP	IF	ID	EX			
BF	IF	--	ID	EX		
转移目标					IF	

先行指令为单精度 FCMP 时，延迟执行 2 个周期。

	←→	←→	←→	←→	←→	←→	槽
FCMP/single	IF	DF	E1	E2			
BF	IF	--	--	ID	EX		
转移目标						IF	

先行指令为双精度 FCMP 时，延迟执行 3 个周期。

	←→	←→	←→	←→	←→	←→	←→	槽
FCMP/double	IF	DF	E1	E1	E2			
BF	IF	--	--	--	ID	EX		
转移目标							IF	

- 指令发行

  该指令使用转移流水线。

- 并行执行可能性

  无特别记载事项。

(2) 带延迟的条件转移指令

• 指令种类

```
BF/S        label
BT/S        label
```

• 流水线

(a) 当条件成立时

	←→	←→	←→	←→	←→	槽	
指令 A	IF	ID	EX				
延迟槽	IF	--	--	ID	EX	…	
下一条指令		IF	…	(取指令，但不执行)			
下下一条指令		IF	…	(取指令，但不执行)			
转移目标指令			--	IF	ID	EX	…

(b) 当条件不成立时

	←→	←→	←→	←→	←→	槽
指令 A	IF	ID	EX			
下一条指令	IF	ID	EX			
下下一条指令		IF	ID	EX	…	
下下下一条指令		IF	ID	EX	…	

• 运行说明

流水线有 IF、 ID、 EX 的 3 个阶段。在 ID 阶段进行条件判断。

在延迟槽不接受中断。

(a) 当条件成立时

EX 阶段中，计算转移目标指令。到此为无效的被取指令全部不执行，转移指令的从指令 A 的 EX 阶段所在槽的下一个槽开始。

(b) 当条件不成立时

如果在 ID 阶段判断出条件不成立，就在 EX 阶段不进行任何运行，而仍继续取下一条指令并执行。典型的流水线如下所示：

先行指令为 CMP 指令时，延迟执行 1 个周期。

	←→	←→	←→	←→	←→	槽
CMP	IF	ID	EX			
BF/S	IF	--	ID	EX		
延迟槽		IF	--	--	ID	

先行指令为单精度 FCMP 时，延迟执行 2 个周期。

	←→	←→	←→	←→	←→	←→	槽
FCMP/single	IF	DF	E1	E2			
BF/S	IF	--	--	ID	EX		
延迟槽		IF	--	--	--	ID	

先行指令为双精度 FCMP 时，延迟执行 3 个周期。

	←→	←→	←→	←→	←→	←→	←→	槽
FCMP/double	IF	DF	E1	E1	E2			
BF/S	IF	--	--	--	ID	EX		
延迟槽		IF	--	--	--	--	ID	

- 指令发行

  该指令使用转移流水线。

  还未对紧接该指令之后的指令 （延迟槽）发行取指令时，该指令延迟执行。

- 并行执行可能性

  无特别记载事项。

(3) 无条件转移指令

• 指令种类

```
BRA     label
BRAF    Rm
BSR     label
BSRF    Rm
JMP     @Rm
JSR     @Rm
RTS
```

• 流水线

	←→	←→	←→	←→	←→	槽
指令 A	IF	ID	EX			
延迟槽	IF	--	--	ID	EX	…
下下一条指令		IF	…	(取指令，但不执行)		
下下下一条指令		IF	…	(取指令，但不执行)		
转移目标指令			--	IF	ID	EX …

• 运行说明

流水线有 IF、 ID、 EX 的 3 个阶段。无条件转移指令为延迟转移指令。

EX 阶段中，计算转移目标地址。无条件转移目标指令 （指令 A）的下一条指令 （即延迟槽指令）如同条件转移指令一样，取指令后执行此指令。但是，此延迟槽指令中 ID 阶段停止 2 个槽的期间。转移指令的从指令 A 的 EX 阶段所在槽的下一个槽开始。

在延迟槽不接受中断。

• 指令发行

该指令使用转移流水线。

还未对紧接该指令之后的指令 （延迟槽）发布取指令时，该指令延迟执行。

• 并行执行可能性

无特别记载事项。

(4) 无延迟的无条件转移指令

• 指令种类

```
JSR/N       @Rm
RTS/N
RTV/N       Rm
```

• 流水线

	←→	←→	←→	←→	←→	槽	
指令 A	IF	ID	EX				
下一条指令	IF	--	…	(取指令，但不执行)			
下下一条指令		IF	…	(取指令，但不执行)			
下下下一条指令		IF	…	(取指令，但不执行)			
转移目标指令			--	IF	ID	EX	…

• 运行说明

流水线有 IF、ID、EX 的 3 个阶段。在 ID 阶段进行条件判断。该转移指令不是延迟转移指令。EX 阶段中计算转移目标指令。到此为无效的被取指令全部不执行。转移指令的从指令 A 的 EX 阶段所在槽的下一个槽开始。

• 指令发行

该指令使用转移流水线。

• 并行执行可能性

无特别记载事项。

(5) 无条件无延迟转移指令 （JSR/N @@ （disp,TBR））

• 指令种类

JSR/N @@（disp,TBR）

• 流水线

	←→	←→	←→	←→	←→	槽			
指令 A	IF	ID	EX	MA	EX				
下一条指令	IF	--	…	(取指令，但不执行)					
下下一条指令		IF	…	(取指令，但不执行)					
下下下一条指令		IF	…	(取指令，但不执行)					
转移目标指令			--	--	--	IF	ID	EX	…

• 运行说明

流水线有 IF、 ID、 EX、 MA、 EX 的 5 个阶段。在 ID 阶段进行条件判断。该转移目标指令不是延迟转移指令。在第二个 EX 阶段计算转移目标地址。到此为无效的被取指令全部不执行。转移指令的从指令 A 的第 2 个 EX 阶段所在槽的下一个槽开始。

• 指令发行

该指令使用转移流水线。

该指令使用存储器存取流水线。

• 并行执行可能性

无特别记载事项。

8.9.6 系统控制指令

(1) 系统控制 ALU 指令

- 指令种类

```
CLRT
LDC     Rm,GBR
LDC     Rm,TBR
LDC     Rm,VBR
LDS     Rm,PR
NOP
SETT
STC     GBR,Rn
STC     TBR,Rn
STC     VBR,Rn
STS     PR,Rn
```

- 流水线

	←→	←→	←→	←→	←→	槽
指令 A	IF	ID	EX			
下一条指令	IF	ID	EX	…		
下下一条指令		IF	ID	EX	…	

- 运行说明

流水线有 IF、 ID、 EX 的 3 个阶段。 EX 阶段中，经过 ALU 完成数据运算。

- 指令发行

该指令不引起资源竞争。

- 并行执行可能性

无特别记载事项。

(2) 系统控制 ALU 指令

• 指令种类

LDC Rm,SR

• 流水线

	←→	←→	←→	←→	←→	←→	←→	槽
指令 A	IF	ID	EX	EX	EX			
下一条指令	IF	--	--	ID	EX	…		
下下一条指令		IF	--	--	ID	EX	…	

• 运行说明

流水线有 IF、 ID、 EX、 EX、 EX 的 5 个阶段。在第 1 个 EX 阶段，经过 ALU 完成数据运算。

• 指令发行

该指令不引起资源竞争。

• 并行执行可能性

该指令为多周期指令，不可与后续指令并行执行。

(3) 系统控制 ALU 指令

• 指令种类

STC SR,Rn

• 流水线

	←→	←→	←→	←→	←→	←→	←→	槽
指令 A	IF	ID	EX	EX				
下一条指令	IF	--	ID	EX	…			
下下一条指令		IF	--	ID	EX	…		

• 运行说明

流水线有 IF、 ID、 EX、 EX 的 4 个阶段。在第 2 个 EX 阶段，经过 ALU 完成数据运算。

• 指令发行

无特别记载事项。

典型的 CS 位读取时的流水线如下所示。

	←→	←→	←→	←→	←→	←→	←→	槽
CLIP	IF	ID	EX					
STC	IF	--	ID	EX	EX			
下一条指令		IF	--	ID	EX	…		
下下一条指令		IF	--	ID	EX	…		

• 并行执行可能性

该指令为多周期指令，不可与后续指令并行执行。

(4) LDC.L、 LDS.L 指令

• 指令种类

```
LDC.L       @Rm+,GBR
LDC.L       @Rm+,VBR
LDS.L       @Rm+,PR
```

• 流水线

	←→	←→	←→	←→	←→	←→	←→	槽
指令 A	IF	ID	EX	MA	WB			
下一条指令	IF	ID	EX	…				
下下一条指令		IF	ID	EX	…			

• 运行说明

流水线有 IF、 ID、 EX、 MA、 WB 的 5 个阶段。

• 指令发行

该指令使用存储器存取流水线。

• 并行执行可能性

无特别记载事项。

(5) LDC.L 指令

• 指令种类

```
LDC.L       @Rm+,SR
```

• 流水线

	←→	←→	←→	←→	←→	←→	←→	槽	
指令 A	IF	ID	EX	MA	EX	EX	EX		
下一条指令	IF	--	--	--	--	ID	EX	…	
下下一条指令		IF	--	--	--	--	ID	EX	…

• 运行说明

流水线有 IF、 ID、 EX、 MA、 EX、 EX、 EX 的 7 个阶段。

• 指令发行

该指令使用存储器存取流水线。

• 并行执行可能性

该指令为多周期指令，不可与后续指令并行执行。

(6) STC.L 指令

• 指令种类

```
STC.L    GBR,@-Rn
STC.L    VBR,@-Rn
STS.L    PR,@-Rn
```

• 流水线

	←→	←→	←→	←→	←→	←→	←→	槽
指令 A	IF	ID	EX	MA				
下一条指令	IF	ID	EX	…				
下下一条指令		IF	ID	EX	…			

• 运行说明

流水线有 IF、 ID、 EX、 MA 的 4 个阶段。

• 指令发行

该指令使用存储器存取流水线。

• 并行执行可能性

无特别记载事项。

(7) STC.L 指令

• 指令种类

```
STC.L    SR,@-Rn
```

• 流水线

	←→	←→	←→	←→	←→	←→	←→	槽
指令 A	IF	ID	EX	EX	MA			
下一条指令	IF	--	ID	EX	…			
下下一条指令		IF	--	ID	EX	…		

• 运行说明

流水线有 IF、 ID、 EX、 EX、 MA 的 5 个阶段。

• 指令发行

该指令使用存储器存取流水线。

• 并行执行可能性

该指令为多周期指令，不可与后续指令并行执行。

另外，该指令使用存储器存取流水线，但单一周期的存储器存取指令为先行指令时可并行执行。

(8) 寄存器 →MAC 传送指令

• 指令种类

```
CLRMAC
LDS        Rm,MACH
LDS        Rm,MACL
```

• 流水线

	←→	←→	←→	←→	←→	←→	←→	槽
指令 A	IF	ID	mm	mm				
下一条指令	IF	ID	EX	…				
下下一条指令		IF	ID	EX	…			

• 运行说明

流水线有 IF、 ID、 mm、 mm 的 4 个阶段。 mm 表示乘法器正在运行的状态。

一般流水线的详细内容，请参照 “8.7 由乘法器引起的竞争”。该指令有 1 个执行槽、 2 个等待时间、 1 个锁槽。关于该指令的流水线、乘法器相关的指令连续时的详细内容例如下所示：

(a) 在紧接着 CLRMAC 指令之后连续出现 MAC.W、 MAC.L 指令时，乘法器不竞争。

	←→	←→	←→	←→	←→	←→	←→	←→	槽
CLRMAC	IF	ID	mm	mm					
MAC.W @Rm+, @Rn+	IF	ID	EX	MA	MA	mm	mm		
下下一条指令		IF	--	--	ID	EX	…		

(b) 在紧接着 CLRMAC 指令之后连续出现 MULS.W、MULU.W、DMULS.L、DMULU.L、MUL.L、MULR、STS （寄存器）、 STS.L （存储器）、 LDS （寄存器）指令时，由于 CLRMAC 指令锁住乘法器，因此不可并行执行。

	←→	←→	←→	←→	←→	←→	←→	←→	槽
CLRMAC	IF	ID	mm	mm					
STS MACL,Rn	IF	--	ID	EX	WB				
下下一条指令		IF	ID	EX	…				

(c) 在紧接着 CLRMAC 指令之后出现 LDS.L （存储器）指令时，延迟执行 CLRMAC 指令的执行状态 （1 个槽）的期间。

←→	←→	←→	←→	←→	←→	←→	←→	←→	槽
CLRMAC	IF	ID	mm	mm					
LDS.L @Rn+,MACL	IF	--	ID	EX	MA	WB			
下下一条指令		IF	ID	EX	…				

• 指令发行

该指令使用乘法器。

该指令锁住乘法器 1 个槽的期间。

• 并行执行可能性

无特别记载事项。

(9) 存储器 →MAC 传送指令

• 指令种类

```
LDS.L    @Rm+,MACH
LDS.L    @Rm+,MACL
```

• 流水线

	←→	←→	←→	←→	←→	槽
指令 A	IF	ID	EX	MA	WB	
下一条指令	IF	ID	EX	…		
下下一条指令		IF	ID	EX	…	

• 运行说明

流水线有 IF、 ID、 EX、 MA、 WB 的 5 个阶段。

一般流水线的详细内容，请参照 “8.7 由乘法器引起的竞争”。该指令有 1 个执行槽、 3 个等待时间、 2 个锁槽。关于该指令的流水线、乘法器相关的指令连续时的详细内容例如下所示：

(a) 在紧接着 LDS.L 指令之后连续出现 MAC.W、 MAC.L 指令时，乘法器不竞争。但是，存储器存取竞争时，停止 1 个周期。

	←→	←→	←→	←→	←→	←→	←→	槽
LDS.L @Rm+, MACH	IF	ID	EX	MA	WB			
MAC.W @Rm+, @Rn+	IF	--	ID	EX	MA	MA	mm	mm
下下一条指令		IF	--	--	ID	EX	…	

(b) 在紧接着 LDS.L 指令之后连续出现 MULS.W、 MULU.W、 DMULS.L、 DMULU.L、 MUL.L、 MULR、 STS （寄存器）、 STS.L （存储器）、 LDS （寄存器）指令时，由于 LDS.L 指令锁住乘法器，因此再停止 1 个槽的期间。

		←→	←→	←→	←→	←→	槽
LDS.L @Rm+, MACH	IF	ID	EX	MA	WB		
STS MACL,Rn	IF	--	--	ID	EX	WB	
下下一条指令		IF	--	ID	EX	…	

(c) 在紧接着 LDS.L 指令之后出现 LDS.L （存储器）指令时，延迟执行 LDS.L 指令的执行状态 （1 个槽）的期间。

	←→	←→	←→	←→	←→	槽
LDS.L @Rn+, MACH	IF	ID	EX	MA	WB	
LDS.L @Rn+,MACL	IF	--	ID	EX	MA	WB
下下一条指令		IF	ID	EX	…	

• 指令发行

该指令使用存储器存取流水线。
该指令使用乘法器。
如果乘法器的锁住期间还剩 1 个槽，则执行该指令。
该指令锁住乘法器 2 个槽的期间。

• 并行执行可能性

无特别记载事项。

(10) MAC→ 寄存器传送指令

• 指令种类

STS	MACH,Rn
STS	MACL,Rn

• 流水线

	←→	←→	←→	←→	←→	槽
指令 A	IF	ID	EX	WB		
下一条指令	IF	ID	EX	…		
下下一条指令		IF	ID	EX	…	

• 运行说明

流水线有 IF、 ID、 EX、 WB 的 4 个阶段。

一般流水线的详细内容，请参照 “8.7 由乘法器引起的竞争”。该指令有 1 个执行槽、 2 个等待时间、 0 个锁槽。关于该指令的流水线、乘法器相关的指令连续时的详细内容例如下所示：

(a) 在紧接着 STS 指令之后连续出现 MAC.W、 MAC.L 指令时，乘法器不竞争。

	←→	←→	←→	←→	←→	←→	←→	槽
STS MACH, Rn	IF	ID	EX	WB				
MAC.W @Rm+, @Rn+	IF	ID	EX	MA	MA	mm	mm	
下下一条指令		IF	--	ID	EX	…		

(b) 在紧接着 STS 指令之后连续出现 MULS.W、MULU.W、DMULS.L、DMULU.L、MUL.L、MULR、STS （寄存器）、 STS.L （存储器）、 LDS （寄存器）指令时，由于 STS 指令不锁住乘法器，因此可并行执行。

	←→	←→	←→	←→	←→	←→	←→	槽
STS MACH, Rn	IF	ID	EX	WB				
MUL.L Rm, Rn	IF	ID	mm	mm	mm			
下下一条指令		IF	ID	EX	…			

(c) 在紧接着 STS 指令之后连续出现 STS （寄存器）、 STS.L （存储器）指令时，由于乘法结果的读取总线竞争，因此不可并行执行。

	←→	←→	←→	←→	←→	←→	←→	槽
STS MACH, Rn	IF	ID	EX	WB				
STS MACL, Rn	IF	--	ID	EX	WB			
下下一条指令		IF	ID	EX	…			

(d) 在紧接着 STS 指令之后出现 LDS.L （存储器）指令时，可并行执行。

	←→	←→	←→	←→	←→	←→	←→	槽
STS MACH, Rn	IF	ID	EX	WB				
LDS.L @Rn+,MACL	IF	ID	EX	MA	WB			
下下一条指令		IF	ID	EX	…			

- 指令发行

  该指令使用乘法器 , 但不锁住。

  该指令使用乘法结果读取总线。

- 并行执行可能性

  无特别记载事项。

(11) MAC→ 存储器传送指令

• 指令种类

```
STS.L    MACH,@-Rn
STS.L    MACL,@-Rn
```

• 流水线

	←→	←→	←→	←→	←→	槽
指令 A	IF	ID	EX	MA		
下一条指令	IF	ID	EX	…		
下下一条指令		IF	ID	EX	…	

• 运行说明

流水线有 IF、 ID、 EX、 MA 的 4 个阶段。

一般流水线的详细内容，请参照 “8.7 由乘法器引起的竞争”。该指令有 1 个执行槽、 2 个等待时间、 0 个锁槽。关于该指令的流水线、乘法器相关的指令连续时的详细内容例如下所示：

(a) 在紧接着 STS.L 指令之后连续出现 MAC.W、 MAC.L 指令时，乘法器不竞争。但是，存储器存取竞争时，停止 1 个周期。

	←→	←→	←→	←→	←→	←→	←→	槽
STS.L MACH, @-Rn	IF	ID	EX	MA				
MAC.W @Rm+, @Rn+	IF	--	ID	EX	MA	MA	mm	mm
下下一条指令		IF	--	--	ID	EX	…	

(b) 在紧接着 STS.L 指令之后连续出现 MULS.W、 MULU.W、 DMULS.L、 DMULU.L、 MUL.L、 MULR、 STS （寄存器）、 STS.L （存储器）、 LDS （寄存器）指令时，由于 STS.L 指令不锁住乘法器，因此可并行执行。

	←→	←→	←→	←→	←→	←→	←→	槽
STS.L MACL, @-Rn	IF	ID	EX	MA				
MUL.L Rm, Rn	IF	ID	mm	mm				
下下一条指令		IF	ID	EX	…			

(c) 在紧接着 STS.L 指令之后连续出现 STS （寄存器）、 STS.L （存储器）时，由于乘法结果读取总线竞争，因此不可并行执行。

	←→	←→	←→	←→	←→	←→	←→	槽
STS.L MACH, @-Rn	IF	ID	EX	MA				
STS.L MACL, @-Rn	IF	--	ID	EX	MA			
下下一条指令		IF	ID	EX	…			

(d) 在紧接着 STS.L 指令之后出现 LDS.L （存储器）指令时，发生存储器存取流水线的竞争，不可并行执行。

	←→	←→	←→	←→	←→	←→	←→	槽
STS.L MACH, @-Rn	IF	ID	EX	MA				
LDS.L @Rn+,MACL	IF	--	ID	EX	MA	WB		
下下一条指令		IF	ID	EX	…			

- 指令发行

  该指令使用存储器存取流水线。
  该指令使用乘法器，但不锁住。
  该指令使用乘法结果读取总线。

- 并行执行可能性

  无特别记载事项。

(12) RTE 指令

- 指令种类

  RTE

- 流水线

	←→	←→	←→	←→	←→	←→	←→	←→	←→	槽
指令 A	IF	ID	EX	MA	MA	EX	EX	EX		
延迟槽	IF	--	--	--	--	--	ID	EX	…	
转移目标						IF	--	ID	EX	…

- 运行说明

流水线有 IF、ID、EX、MA、MA、EX、EX、EX 的 8 个阶段。RTE 为延迟转移目标指令。延迟槽指令的 ID 停止 5 个槽期间。转移指令的 IF 从 RTE 的第 2 个 MA 的下一个槽开始。

- 指令发行

  该指令不引起资源竞争。

- 并行执行可能性

  该指令为多周期指令，不可与后续指令并行执行。

(13) RESBANK 指令

• 指令种类

RESBANK

• 流水线

○ BO==0 时

	←→	←→	←→	←→	←→	←→	←→	←→	←→	←→	←→	槽
指令 A	IF	ID	EX	EX	EX	EX	EX	EX	EX	EX	EX	
下一条指令	IF	--	--	--	--	--	--	--	--	ID	EX	…

○ BO==1 时

	←→	←→	←→	←→	←→	←→	…	←→	←→	←→	槽
指令 A	IF	ID	EX	MA	MA	MA	…	MA	MA	MA	WB
下一条指令	IF	--	--	--	--	--	…	ID	EX	…	

• 运行说明

BO 位 =0 时与 BO 位 =1 时的运行不同。

BO 位 =0 时，进行从存储体的返回。流水线有 IF、ID、EX、EX、EX、EX、EX、EX、EX、EX、EX（重复运行 9 次 EX）的 11 个阶段。在此期间，寄存器从存储体返回。

BO 位 =1 时，进行从堆栈的返回。流水线有 IF、ID、EX、MA、MA、MA、MA、MA、MA、MA、MA、MA、MA、MA、MA、MA、MA、MA、MA、MA、MA（重复运行 19 次 MA）、WB 的 23 个阶段。

• 指令发行

BO 位 =0 时，该指令不引起资源竞争。

BO 位 =1 时，该指令使用存储器存取流水线。

• 并行执行可能性

该指令为多周期指令，不可与后续指令并行执行。（参照 “8.3.4 由多周期指令引起的竞争的详细内容”）。

(14) LDBANK 指令

• 指令种类

```
LDBANK @Rm,R0
```

• 流水线

	←→	←→	←→	←→	←→	←→	…	←→	槽	
指令 A	IF	ID	EX	EX	EX	EX	EX	EX		
下一条指令	IF	--	--	--	--	--	ID	EX	…	
下下一条指令		IF	--	--	--	--	--	ID	EX	…

• 运行说明

流水线有 IF、 ID、 EX、 EX、 EX、 EX、 EX、 EX 的 8 个阶段。

• 指令发行

该指令，不引起资源竞争。

• 并行执行可能性

该指令为多周期指令，不可与后续指令并行执行。（参照 “8.3.4 由多周期指令引起的竞争的详细内容”）。

(15) STBANK 指令

• 指令种类

```
STBANK R0,@Rn
```

• 流水线

	←→	←→	←→	←→	←→	←→	←→	←→	←→	槽	
指令 A	IF	ID	EX	EX	EX	EX	EX	EX	EX		
下一条指令	IF	--	--	--	--	--	--	ID	EX	…	
下下一条指令		IF	--	--	--	--	--	--	ID	EX	…

• 运行说明

流水线有 IF、 ID、 EX、 EX、 EX、 EX、 EX、 EX、 EX 的 9 个阶段。

• 指令发行

该指令不引起资源竞争。

• 并行执行可能性

该指令为多周期指令，不可与后续指令并行执行。（参照 “8.3.4 由多周期指令引起的竞争的详细内容”）。

(16) TRAP 指令

- 指令种类

  TRAPA #imm

- 流水线

	←→	←→	←→	←→	←→	←→	←→	←→	槽
指令 A	IF	ID	EX	EX	EX	MA	MA	MA	
下一条指令	IF	--	…						
下下一条指令		IF	--	…					
转移目标								IF	

- 运行说明

  流水线有 IF、ID、EX、EX、EX、MA、MA、MA 的 8 个阶段。TRAP 指令不是延迟转移指令。转移目标指令的 IF 从 TRAP 指令的第 3 个 MA 所在的槽开始。

- 指令发行

  该指令使用存储器存取流水线。

- 并行执行可能性

  该指令为周期指令，不可与后续指令并行执行。

(17) SLEEP 指令

- 指令种类

  SLEEP

- 流水线

	←→	←→	←→	←→	←→	←→	←→	←→	←→	槽
指令 A	IF	ID	EX	MA	EX	EX	EX			
下一条指令	IF	--	…							
下下一条指令		IF	--	…						

- 运行说明

  流水线有 IF、 ID、 EX、 MA、 EX、 EX、 EX 的 7 个阶段。
  在执行 SLEEP 指令后进入睡眠模式或者待机模式。

- 指令发行

  该指令使用存储器存取流水线。

- 并行执行可能性

  该指令为多周期指令，不可与后续指令并行执行。

8.9.7 异常处理

(1) 中断异常处理

- 指令种类

  中断异常处理

- 流水线

  ○无堆积 （banking）时

	←→	←→	←→	←→	←→	←→	←→	←→	槽
中断	IF	ID	EX	EX	MA	MA	MA		
下一条指令	IF	…							
下下一条指令		IF	…						
转移目标							IF	ID	

  ○有堆积 （banking）并且无上溢时

	←→	←→	←→	←→	←→	←→	←→	←→	槽
中断	IF	ID	EX	EX	MA	MA	MA	EX	
下一条指令	IF	…							
转移目标							IF	ID	

  ○有堆积 （banking）并且上溢时

	←→	←→	←→	←→	←→	←→	←→	←→	←→	槽
中断	IF	ID	EX	EX	MA	MA	MA	…	MA	
下一条指令	IF	…								
转移目标							IF			… ID

- 运行说明

  在指令的 ID 阶段接受中断，将该 ID 阶段以后置换为中断异常处理顺序。

  在无堆积 （banking）、有堆积 （banking）、有堆积 （banking）并且上溢时中断处理运行各不相同。

  无堆积 （banking）时，流水线有 IF、 ID、 EX、 EX、 MA、 MA、 MA 的 7 个阶段。

  有堆积 （banking）并且无上溢时，自动进行向存储体的保存。流水线有 IF、 ID、 EX、 EX、 MA、 MA、 MA、 EX 的 8 个阶段。

  有堆积 （banking）并且上溢时，将向存储体保存的寄存器自动保存到堆栈，并将 BO 位置 1。此时，流水线有 IF、ID、EX、EX、MA、MA、MA、EX、MA、MA、MA、MA、MA、MA、MA、MA、MA、MA、MA、MA、MA、MA、MA、MA、MA、MA、MA 的 27 个阶段，其中重复运行 2 次 EX、 3 次 MA、 1 次 EX、 19 次 MA。

  中断异常处理不是延迟转移。转移目标指令的从中断异常处理的第 3 个 MA 所在的槽开始。

  中断源有由 NMI 等外部中断请求引脚、用户断点、内部外围模块产生的中断。

- 中断接受

  在延迟槽不可接受中断异常处理。

  有目前正在执行中的多周期指令时，不接受中断异常处理。在该指令执行完成之后可接受中断异常处理。但是在执行中取消 DIVU、 DIVS 指令，就可接受中断。

(2) 地址错误异常处理

• 指令种类

地址错误异常处理

• 流水线

	←→	←→	←→	←→	←→	←→	←→	槽
地址错误异常处理	IF	ID	EX	EX	MA	MA	MA	
下一条指令	IF	…						
下下一条指令		IF	…					
转移目标							IF	ID

• 运行说明

指令的 ID 阶段可接受地址错误，将该 ID 阶段之后置换为地址错误异常处理顺序。

流水线有 IF、ID、EX、EX、MA、MA、MA 的 7 个阶段。地址错误异常处理不是延迟转移。转移目标指令的从地址错误异常处理最后的 MA 阶段所在的槽开始。

地址错误的发生源有取指令引起的发生源或者读写数据引起的发生源。有关发生源的详细内容请参照各硬件手册。

• 地址异常处理接受

在延迟槽不可接受地址异常处理。

有目前正在执行的多周期指令时，不可接受中断异常处理。在该指令执行完成之后可接受中断异常处理。但是执行中取消 DIVU、 DIVS 指令，就可接受地址异常处理。

(3) 非法指令异常处理

• 指令种类

非法指令异常处理

• 流水线

	←→	←→	←→	←→	←→	←→	←→	←→	←→	←→	槽
非法指令	IF	ID	EX	EX	MA	MA	MA				
下一条指令	IF	--	…								
下下一条指令		IF	--	…							
转移目标							IF	ID			

• 运行说明

在指令的 ID 阶段可接受非法指令，将该 ID 阶段以后置换为非法指令异常处理顺序。流水线有 IF、 ID、 EX、 EX、 MA、 MA、 MA 的 7 个阶段。非法指令异常处理不是延迟转移。

有通过一般非法指令以及槽非法指令产生的非法指令异常处理源。解码配置在紧接着延迟转移指令之后的槽（称为延迟槽）以外的未定义码时，为一般非法指令异常处理；解码配置在延迟槽内的未定义码或将改写程序计数器的指令、 32 位指令、 RESBANK 指令以及 DIVU、 DIVS 指令配置在延迟槽，并将其进行解码时，为槽非法指令的异常处理。

如果在 FPU 模块停止时执行 FPU 指令及 FPU 相关的 CPU 指令时，为一般非法指令异常处理。

转移目标指令的从非法指令异常处理最后的 MA 阶段所在的槽开始。

(4) FPU 异常处理

• 指令种类

FPU 异常处理

• 流水线

	←→	←→	←→	←→	←→	←→	←→	槽
FPU 异常处理	IF	ID	EX	EX	MA	MA	MA	
下一条指令	IF	…						
下下一条指令		IF	…					
转移目标							IF	ID

• 运行说明

在指令的 ID 阶段可接受 FPU 异常处理，将该 ID 阶段以后置换为 FPU 异常处理顺序。

流水线有 IF、ID、EX、EX、MA、MA、MA 的 7 个阶段。FPU 异常处理不是延迟转移。转移目标指令从 FPU 异常处理的最后 MA 阶段所在的槽开始。

• 关于从发生 FPU 异常到接受 FPU 异常为止进行流水线处理的指令

FPU 不仅将产生异常的指令 NOP 化，而且将发生异常后到接受异常指令期间的 FPU 指令 （除 FCMP 指令） NOP 化。因此，不进行通过该区间指令更新 FPU 寄存器。

FPU 相关的 CPU 指令，与上述内容相同， FPU 的寄存器不更新 （NOP 化），但 CPU 的寄存器更新。

CPU 指令不 NOP 化，按照平常那样运行。

8.9.8 浮点指令及 FPU 相关的 CPU 指令

(1) FPUL 加载指令

- 指令种类

```
LDS       Rm,FPUL
LDS.L     @Rm+,FPUL
```

- 流水线

	←→	←→	←→	←→	←→	←→	←→	槽
指令 A	IF	ID	EX	MA				：CPU 流水线
	IF	DF	EX	NA	SF			：FPU 流水线
下一条指令	IF	ID	EX	…				：CPU 流水线
	IF	DF	…					：FPU 流水线
下下一条指令		IF	ID	EX	…			：CPU 流水线
		IF	DF	…				：FPU 流水线

- 运行说明

流水线中，CPU 流水线有 IF、ID、EX、MA 的 4 个阶段，FPU 流水线有 IF、DF、EX、NA、SF 的 5 阶段。在该指令的 1 ～ 3 条指令后设置读 FPUL 的指令时，可能会发生竞争。

- 指令发行

该指令使用 FPU 加载存储流水线与存储器存取流水线。

但是 LDS 指令与 CPU 存储器读指令不发生竞争。

- 并行执行可能性

无特别记载事项。

(2) FPSCR 加载指令

• 指令种类

```
LDS       Rm,FPSCR
LDS.L     @Rm+,FPSCR
```

• 流水线

	←→	←→	←→	←→	←→	←→	←→	槽
指令 A	IF	ID	EX	MA				：CPU 流水线
	IF	DF	EX	NA	SF			：FPU 流水线
下一条指令	IF	--	ID	EX	…			：CPU 流水线
	IF	--	DF	…				：FPU 流水线
下下一条指令		IF	ID	EX	…			：CPU 流水线
		IF	DF	…				：FPU 流水线

• 运行说明

流水线中，CPU 流水线有 IF、ID、EX、MA 的 4 个阶段； FPU 流水线有 IF、DF、EX、NA、SF 的 5 阶段。后续的 FPU 相关的指令，停止 3 个周期。

• 指令发行

该指令使用 FPU 加载储存流水线。

LDS.L 指令也使用存储器存取流水线。

有运算中的 FPU 算术运算指令时，等待到结束。

• 并行执行可能性

该指令不可与 FPU 指令及 FPU 相关的 CPU 指令并行执行。

(3) FPUL 存储指令 （STS）

• 指令种类

STS FPUL,Rn

• 流水线

	←→	←→	←→	←→	←→	←→	←→ 槽	
指令 A	IF	ID	EX	WB				：CPU 流水线
	IF	DF	EX	NA				：FPU 流水线
下一条指令	IF	ID	EX	…				：CPU 流水线
	IF	DF	…					：FPU 流水线
下下一条指令		IF	ID	EX	…			：CPU 流水线
		IF	DF	…				：FPU 流水线

• 运行说明

流水线中，CPU 流水线有 IF、ID、EX、WB 的 4 个阶段，而 FPU 流水线有 IF、DF、EX、NA 的 4 个阶段。如果将使用该指令的目标的指令设置在该指令的 1 ～ 3 条指令之后，可能会发生竞争。

• 指令发行

该指令使用乘法结果读取总线。

该指令使用 FPU 加载存储流水线。

FPUL 等待 FPU 算术运算结果时，之前指令的等待时间数减少 2。详细内容请参照 “8.6 由 FPU 引起的竞争”。

• 并行执行可能性

无特别记载事项。

(4) FPUL 存储指令 （STS.L）

• 指令种类

STS.L FPUL,@-Rn

• 流水线

	←→	←→	←→	←→	←→	←→	←→	槽
指令 A	IF	ID	EX	MA				：CPU 流水线
	IF	DF	EX	NA				：FPU 流水线
下一条指令	IF	ID	EX	…				：CPU 流水线
	IF	DF	…					：FPU 流水线
下下一条指令		IF	ID	EX	…			：CPU 流水线
		IF	DF	…				：FPU 流水线

• 运行说明

流水线中，CPU 流水线有 IF、ID、EX、MA 的 4 个阶段，而 FPU 流水线有 IF、DF、EX、NA 的 4 个阶段。

• 指令发行

该指令使用 FPU 加载存储流水线与存储器存取流水线。

FPUL 等待 FPU 算术运算结果时，之前的指令的等待时间数减少 1。

详细内容请参照 “8.6 由 FPU 引起的竞争”

• 并行执行可能性

无特别记载事项。

(5) FPSCR 存储指令 （STS）

• 指令种类

STS FPSCR, Rn

• 流水线

	←→	←→	←→	←→	←→	←→	←→	槽
指令 A	IF	ID	EX	WB				：CPU 流水线
	IF	DF	EX	NA				：FPU 流水线
下一条指令	IF	--	ID	EX	…			：CPU 流水线
	IF	--	DF	…				：FPU 流水线
下下一条指令		IF	ID	EX				：CPU 流水线
		IF	DF	…				：FPU 流水线

• 运行说明

流水线中，CPU 流水线有 IF、ID、EX、WB 的 4 个阶段，而 FPU 流水线有 IF、DF、EX、NA 的 4 个阶段。

如果将使用该指令的目标的指令设置在该指令的 1 ～ 3 条指令之后，可能会发生竞争。

• 指令发行

该指令使用乘法结果读取总线。

该指令使用 FPU 加载存储流水线与存储器存取流水线。

有运算中的 FPU 算术运算指令时，等待到结束。

• 并行执行可能性

该指令不可与 FPU 指令及 FPU 相关的 CPU 指令并行执行。

(6) FPSCR 存储指令 （STS.L）

• 指令种类

```
STS.L       FPSCR,@-Rn
```

• 流水线

	←→	←→	←→	←→	←→	←→	←→	槽
指令 A	IF	ID	EX	MA			：CPU 流水线	
	IF	DF	EX	NA			：FPU 流水线	
下一条指令	IF	--	ID	EX	…		：CPU 流水线	
	IF	--	DF	…			：FPU 流水线	
下下一条指令		IF	ID	EX	…		：CPU 流水线	
		IF	DF	…			：FPU 流水线	

• 运行说明

流水线中，CPU 流水线有 IF、ID、EX、MA 的 4 个阶段，而 FPU 流水线有 IF、DF、EX、NA 的 4 个阶段。

• 指令发行

该指令使用 FPU 加载存储流水线与存储器存取流水线。

有运算中的 FPU 算术运算指令时，等待到结束。

• 并行执行可能性

该指令不可与 FPU 指令及 FPU 相关的 CPU 指令并行执行。

(7) 浮点寄存器－寄存器间传送指令 / 浮点寄存器－立即指令 / 浮点运算指令的一部分

- 指令种类

```
FLDS        FRm,FPUL
FMOV        FRm,FRn
FSTS        FPUL,FRn
FLDI0       FRn
FLDI1       FRn
FABS        FRn
FNEG        FRn
FABS        DRn
FNEG        DRn
```

- 流水线

	←→	←→	←→	←→	←→	←→	←→	槽
指令 A	IF	ID	EX				：CPU 流水线	
	IF	DF	EX	NA	SF		：FPU 流水线	
下一条指令	IF	ID	EX	…			：CPU 流水线	
	IF	DF	E1	E2	SF		：FPU 流水线	
下下一条指令		IF	ID	EX	…		：CPU 流水线	
		IF	DF	E1	E2	E3	：FPU 流水线	

- 运行说明

流水线中， CPU 流水线有 IF、 ID、 EX 的 3 个阶段，而 FPU 流水线有 IF、 DF、 EX、 NA、 SF 的 5 个阶段。即使将读该指令的目标的指令设置在紧接该指令之后，也不会发生竞争。

- 指令发行

该指令使用 FPU 加载存储流水线。

- 并行执行可能性

该指令为 0 个等待时间的指令。该指令作为先行指令执行并且后续指令使用 FRn、 FPUL 时也可并行执行。

(8) 双精度浮点寄存器－寄存器间传送指令

• 指令种类

```
FMOV        DRm,DRn
```

• 流水线

	←→	←→	←→	←→	←→	←→	←→	槽
指令 A	IF	ID	EX	EX				：CPU 流水线
	IF	DF	EX	EX	NA	SF		：FPU 流水线
下一条指令	IF	…	ID	EX	…			：CPU 流水线
	IF	…	DF	E1	E2	SF		：FPU 流水线
下下一条指令		IF	…	ID	EX	…		：CPU 流水线
		IF	…	DF	E1	E2	SF	：FPU 流水线

• 运行说明

流水线中，CPU 流水线有 IF、ID、EX、EX 的 4 个阶段，而 FPU 流水线有 IF、DF、EX、EX、NA、SF 的 6 个阶段。

• 指令发行

该指令使用 FPU 加载存储流水线。

• 并行执行可能性

无特别记载事项。

(9) FSCHG 指令

• 指令种类

```
FSCHG
```

• 流水线

	←→	←→	←→	←→	←→	←→	←→	槽
指令 A	IF	ID	EX				：CPU 流水线	
	IF	DF	EX	NA	SF		：FPU 流水线	
下一条指令	IF	ID	EX	…			：CPU 流水线	
	IF	DF	E1	E2	SF		：FPU 流水线	
下下一条指令		IF	ID	EX	…		：CPU 流水线	
		IF	DF	E1	E2	SF	：FPU 流水线	

• 运行说明

流水线中，CPU 流水线有 IF、ID、EX 的 3 个阶段，而 FPU 流水线有 IF、DF、EX、NA、SF 的 5 个阶段。即使将读该指令的目标的指令设置在紧接该指令之后，也不会发生竞争。

• 指令发行

该指令使用 FPU 加载存储流水线。

• 并行执行可能性

无特别记载事项。

(10) 浮点寄存器的装入指令

• 指令种类

```
FMOV.S      @Rm,FRn
FMOV.S      @Rm+,FRn
FMOV.S      @(R0,Rm),FRn
FMOV.D      @Rm,DRn
FMOV.D      @Rm+,DRn
FMOV.D      @(R0,Rm),DRn
```

• 流水线 （单精度）

	←→	←→	←→	←→	←→	←→	←→	槽
指令 A	IF	ID	EX	MA				：CPU 流水线
	IF	DF	EX	NA	SF			：FPU 流水线
下一条指令	IF	ID	EX	…				：CPU 流水线
	IF	DF	E1	E2	SF			：FPU 流水线
下下一条指令		IF	ID	EX	…			：CPU 流水线
		IF	DF	E1	E2	SF		：FPU 流水线

• 流水线 （双精度）

	←→	←→	←→	←→	←→	←→	←→	槽
指令 A	IF	ID	EX	MA	MA			：CPU 流水线
	IF	DF	EX	EX	NA	SF		：FPU 流水线
下一条指令	IF	--	ID	EX	…			：CPU 流水线
	IF	--	DF	E1	E2	SF		：FPU 流水线
下下一条指令		IF	--	ID	EX	…		：CPU 流水线
		IF	--	DF	E1	E2	SF	：FPU 流水线

• 运行说明 （单精度）

流水线中， CPU 流水线有 IF、 ID、 EX、 MA 的 4 个阶段，而 FPU 流水线有 IF、 DF、 EX、 NA、 SF 的 5 个阶段。如果将读该指令的目标的指令设置在该指令的 1 ～ 3 条指令之后，可能会发生竞争。

• 运行说明 （双精度）

流水线中， CPU 流水线有 IF、 ID、 EX、 MA、 MA 的 5 个阶段，而 FPU 流水线有 IF、 DF、 EX、 EX、 NA、 SF 的 6 个阶段。如果将读该指令的目标的指令设置在该指令的 1 ～ 5 条指令之后，可能会发生竞争。

• 指令发行

该指令使用 FPU 加载存储流水线与存储器存取流水线。

• 并行执行可能性

FMOV.D 指令为多周期指令，不可与后续指令并行执行 （参照 “8.3.4 由多周期指令引起的竞争的详细内容”）。

(11) 浮点寄存器加载指令 （12 位位移量）

• 指令种类

```
FMOV.S      @(disp12,Rm),FRn
FMOV.D      @(disp12,Rm),DRn
```

• 流水线 （单精度）

	←→	←→	←→	←→	←→	←→	←→	槽
指令 A	IF	ID	EX	MA				：CPU 流水线
	IF	DF	EX	NA	SF			：FPU 流水线
下一条指令		IF	ID	EX	…			：CPU 流水线
		IF	DF	EX	NA	SF		：FPU 流水线
下下一条指令		IF	ID	EX	…			：CPU 流水线
		IF	DF	E1	E2	SF		：FPU 流水线

• 流水线 （双精度）

	←→	←→	←→	←→	←→	←→	←→	←→	槽
指令 A	IF	ID	EX	MA	MA				：CPU 流水线
	IF	DF	EX	EX	NA	SF			：FPU 流水线
下一条指令		IF	--	ID	EX	…	…		：CPU 流水线
		IF	--	DF	E1	E2	SF		：FPU 流水线
下下一条指令			IF	--	ID	EX	…		：CPU 流水线
			IF	--	DF	E1	E2	SF	：FPU 流水线

• 运行说明 （单精度）

流水线中，CPU 流水线有 IF、ID、EX、MA 的 4 个阶段，而 FPU 流水线有 IF、DF、EX、NA、SF 的 5 个阶段。如果将读该指令的目标的指令设置在该指令的 1 ～ 3 条指令之后，可能会发生竞争。

• 运行说明 （双精度）

流水线中， CPU 流水线有 IF、 ID、 EX、 MA、 MA 的 5 个阶段，而 FPU 流水线有 IF、 DF、 EX、 EX、 NA、 SF 的 6 个阶段。如果将读该指令的目标的指令设置在该指令的 1 ～ 3 条指令之后，可能会发生竞争。

• 指令发行

该指令使用 FPU 加载存储流水线与存储器存取流水线。

• 并行执行可能性

该指令为 32 位指令，不可并行执行 （参照 “8.3.5 由 32 位指令引起的竞争的详细内容”）。

(12) 浮点寄存器存储指令

• 指令种类

```
FMOV.S      FRm,@Rn
FMOV.S      FRm,@-Rn
FMOV.S      FRm,@(R0,Rn)
FMOV.D      DRm,@Rn
FMOV.D      DRm,@-Rn
FMOV.D      DRm,@(R0,Rn)
```

• 流水线 （单精度）

	←→	←→	←→	←→	←→	←→	←→	槽
指令 A	IF	ID	EX	MA				：CPU 流水线
	IF	DF	EX	NA				：FPU 流水线
下一条指令	IF	ID	EX	…				：CPU 流水线
	IF	DF	E1	E2	SF			：FPU 流水线
下下一条指令		IF	ID	EX	…			：CPU 流水线
		IF	DF	E1	E2	SF		：FPU 流水线

• 流水线 （双精度）

	←→	←→	←→	←→	←→	←→	←→	槽
指令 A	IF	ID	EX	MA	MA			：CPU 流水线
	IF	DF	EX	EX	NA			：FPU 流水线
下一条指令	IF	--	ID	EX	…			：CPU 流水线
	IF	--	DF	E1	E2	SF		：FPU 流水线
下下一条指令		IF	--	ID	EX	…		：CPU 流水线
		IF	--	DF	E1	E2	SF	：FPU 流水线

• 运行说明 （单精度）

流水线中， CPU 流水线有 IF、 ID、 EX、 MA 的 4 个阶段 , 而 FPU 流水线有 IF、 DF、 EX、 NA 的 4 个阶段。

• 运行说明 （双精度）

流水线中， CPU 流水线有 IF、 ID、 EX、 MA、 MA 的 5 个阶段 , 而 FPU 流水线有 IF、 DF、 EX、 EX、 NA 的 5 个阶段。

• 指令发行

该指令使用 FPU 加载存储流水线与存储器存取流水线。

• 并行执行可能性

FMOV.D 指令为多周期指令，不可与后续指令并行执行 （参照 “8.3.4 由多周期指令引起的竞争的详细内容”）。

(13) 浮点寄存器存储指令 （12 位位移量）

• 指令种类

```
FMOV.S      FRm,@(disp12,Rn)
FMOV.D      DRm,@(disp12,Rn)
```

• 流水线 （单精度）

	←→	←→	←→	←→	←→	←→	←→	槽
指令 A	IF	ID	EX	MA				：CPU 流水线
	IF	DF	EX	NA				：FPU 流水线
下一条指令		IF	ID	EX	…			：CPU 流水线
		IF	DF	E1	E2	SF		：FPU 流水线
下下一条指令		IF	ID	EX	…			：CPU 流水线
		IF	DF	EX	NA	SF		：FPU 流水线

• 流水线 （双精度）

	←→	←→	←→	←→	←→	←→	←→	←→	槽
指令 A		IF	ID	EX	MA	MA			：CPU 流水线
		IF	DF	EX	EX	NA			：FPU 流水线
下一条指令		IF	--	ID	EX	…			：CPU 流水线
		IF	--	DF	E1	E2	SF		：FPU 流水线
下下一条指令			IF	--	ID	EX	…		：CPU 流水线
			IF	--	DF	E1	E2	SF	：FPU 流水线

• 运行说明 （单精度）

流水线中， CPU 流水线有 IF、 ID、 EX、 MA 的 4 个阶段 , 而 FPU 流水线有 IF、 DF、 EX、 NA 的 4 个阶段。

• 运行说明 （双精度）

流水线中，CPU 流水线有 IF、ID、EX、MA、MA 的 5 个阶段 , 而 FPU 流水线有 IF、DF、EX、EX、NA 的 5 个阶段。

• 指令发行

该指令使用 FPU 加载存储流水线与存储器存取流水线。

• 并行执行可能性

该指令为 32 位指令，不可并行执行 （参照 “8.3.5 由 32 位指令引起的竞争的详细内容”）。

(14) 浮点运算指令 （FDIV、 FSQRT、 FLOAT、 FTRC 以外）

• 指令种类

```
FADD        FRm,FRn
FMAC        FR0,FRm,FRn
FMUL        FRm,FRn
FSUB        FRm,FRn
FADD        DRm,DRn
FMUL        DRm,DRn
FSUB        DRm,DRn
```

• 流水线 （单精度）

	←→	←→	←→	←→	←→	←→	←→	槽
指令 A	IF	ID	EX				：CPU 流水线	
	IF	DF	E1	E2	SF		：FPU 流水线	
下一条指令	IF	ID	EX	…			：CPU 流水线	
	IF	DF	EX	NA	SF		：FPU 流水线	
下下一条指令		IF	ID	EX	…		：CPU 流水线	
		IF	DF	EX	NA	SF	：FPU 流水线	

• 流水线 （双精度）

	←→	←→	←→	←→	←→	←→	←→	←→	←→	←→	←→	槽
指令 A	IF	ID	EX								：CPU 流水线	
	IF	DF	E1	E1	E1	E1	E1	E1	E2	SF	：FPU 流水线	
下一条指令	IF	ID	EX	MA	WB						：CPU 流水线	
	IF	DF	EX	NA	SF						：FPU 流水线	
下下一条指令		IF	ID	EX	…						：CPU 流水线	
		IF	DF	EX	NA	SF					：FPU 流水线	

• 运行说明 （单精度）

流水线中，CPU 流水线有 IF、ID、EX 的 3 个阶段 , 而 FPU 流水线有 IF、DF、E1、E2、SF 的 5 个阶段。如果将读该指令的目标的指令设置在该指令的 1 ～ 5 条指令之后，可能会发生竞争。

• 运行说明 （双精度）

流水线中， CPU 流水线有 IF、ID、EX 的 3 个阶段 , 而 FPU 流水线有 IF、DF、E1、E1、E1、E1、E1、E1、 E2、 SF 的 10 个阶段。如果将读该指令的目标的指令设置在该指令的 1 ～ 15 条指令之后，可能会发生竞争。

• 指令发行

该指令使用 FPU 算术运算流水线。关于竞争的详细内容请参照 “8.6 由 FPU 引起的竞争”。

• 并行执行可能性

无特别记载事项。

(15) 浮点运算指令 （FLOAT、 FTRC）与 FCNVSD、 FCNVDS 指令

• 指令种类

```
FLOAT     FPUL,FRn
FTRC      DRm,FPUL
FLOAT     FPUL,DRn
FTRC      DRm,FPUL
FCNVSD
FCNVDS
```

• 流水线 （单精度）

	←→	←→	←→	←→	←→	←→	←→	槽
指令 A	IF	ID	EX				：CPU 流水线	
	IF	DF	E1	E2	SF		：FPU 流水线	
下一条指令	IF	ID	EX	…			：CPU 流水线	
	IF	DF	EX	NA	SF		：FPU 流水线	
下下一条指令		IF	ID	EX	…		：CPU 流水线	
		IF	DF	E1	E2	SF	：FPU 流水线	

• 流水线 （双精度）

	←→	←→	←→	←→	←→	←→	←→	槽
指令 A	IF	ID	EX				：CPU 流水线	
	IF	DF	E1	E1	E2	SF	：FPU 流水线	
下一条指令	IF	ID	EX	MA	WB		：CPU 流水线	
	IF	DF	EX	NA	SF		：FPU 流水线	
下下一条指令		IF	ID	EX	…		：CPU 流水线	
		IF	DF	EX	NA	SF	：FPU 流水线	

• 运行说明 （单精度）

流水线中，CPU 流水线有 IF、ID、EX 的 3 个阶段 , 而 FPU 流水线有 IF、DF、E1、E2、SF 的 5 个阶段。如果将读该指令的目标的指令设置在该指令的 1 ～ 5 条指令之后，可能会发生竞争。

• 运行说明 （双精度）

流水线中，CPU 流水线有 IF、ID、EX 的 3 个阶段 , 而 FPU 流水线有 IF、DF、E1、E1、E2、SF 的 6 个阶段。如果将读该指令的目标的指令设置在该指令的 1 ～ 7 条指令之后，可能会发生竞争。

• 指令发行

该指令使用 FPU 算术运算流水线。关于竞争的详细内容请参照 “8.6 由 FPU 引起的竞争”。

• 并行执行可能性

无特别记载事项。

(16) 浮点运算指令 （FDIV）

• 指令种类

```
FDIV        FRm,FRn
FDIV        DRm,DRn
```

• 流水线 （单精度）

	←→	←→	←→	←→	←→	←→	←→	←→	←→	←→	←→	←→	←→	←→	槽
指令 A	IF	ID	EX												：CPU 流水线
	IF	DF	E1	ED	ED	ED	ED	ED	ED	ED	ED	E1	E2	SF	：FPU 流水线
下一条指令	IF	ID	EX	…											：CPU 流水线
	IF	DF	EX	NA	SF										：FPU 流水线
下下一条指令		IF	ID	EX	…										：CPU 流水线
		IF	DF	EX	NA	SF									：FPU 流水线

• 流水线 （双精度）

	←→	←→	←→	←→	←→	←→	←→	←→	←→	←→	←→	←→	←→	←→	槽
指令 A	IF	ID	EX												：CPU 流水线
	IF	DF	E1	E1	ED	…	ED	E1	E1	E1	E2	SF			：FPU 流水线
下一条指令	IF	ID	EX	…											：CPU 流水线
	IF	DF	EX	NA	SF										：FPU 流水线
下下一条指令		IF	ID	EX	…										：CPU 流水线
		IF	DF	EX	NA	SF									：FPU 流水线

- 运行说明 （单精度）

流水线中，CPU 流水线有 IF、ID、EX 的 3 个阶段 , 而 FPU 流水线有 IF、DF、E1、ED、ED、ED、ED、ED、ED、ED、ED、E1、E2、SF 的 14 个阶段。即：进行 1 次 E1 阶段后，重复 8 次 ED 阶段，最后依次进入 E1、 E2、 SF 阶段。

- 运行说明 （双精度）

流水线中，CPU 流水线有 IF、ID、EX 的 3 个阶段 , 而 FPU 流水线有 IF、DF、E1、E1、ED、ED、ED、ED、ED、ED、ED、ED、ED、ED、ED、ED、ED、ED、ED、ED、ED、ED、E1、E1、E1、E2、SF 的 27 阶段。即：进行 2 次 E1 阶段后，重复 18 次 ED 阶段，最后依次进入 E1、 E1、 E1、 E2、 SF 阶段。

发生如 “8.6 由 FPU 引起的竞争”所示的竞争。 FDIV 流水线中， FDIV 结果寄存器存取的指令发生重叠时，该指令等待到 FDIV 指令执行结束。 E1 以后的阶段停止到 FDIV 的执行结束后，以后的指令也停止。因此，如果在从紧接着 FDIV 指令之后到单精度时为 21 条、双精度时为 49 条指令以内，不配置 FDIV 结果寄存器的浮点指令或 FPU 相关的 CPU 指令，在此期间就可执行 CPU 指令或其他 FPU 指令，提高其性能。

- 指令发行

该指令使用 FPU 算术运算流水线。关于竞争的详细内容请参照 “8.6 由 FPU 引起的竞争”。

该指令的 ED 阶段与槽无关，以状态运行。

- 并行执行可能性

无特别记载事项。

(17) 浮点运算指令 （FSQRT）

• 指令种类

FSQRT FRn

FSQRT DRn

• 流水线 （单精度）

	←→	←→	←→	←→	←→	←→	←→	←→	←→	←→	←→	←→	←→	←→ 槽
指令 A	IF	ID	EX											：CPU 流水线
	IF	DF	E1	ED	ED	ED	ED	ED	ED	ED	E1	E2	SF	：FPU 流水线
下一条指令	IF	ID	EX	…										：CPU 流水线
	IF	DF	EX	NA	SF									：FPU 流水线
下下一条指令		IF	ID	EX	…									：CPU 流水线
		IF	DF	EX	NA	SF								：FPU 流水线

• 流水线 （双精度）

	←→	←→	←→	←→	←→	←→	←→	←→	←→	←→	←→	←→	←→	←→ 槽
指令 A	IF	ID	EX											：CPU 流水线
	IF	DF	E1	E1	ED	…	ED	E1	E1	E1	E2	SF		：FPU 流水线
下一条指令	IF	ID	EX	…										：CPU 流水线
	IF	DF	EX	NA	SF									：FPU 流水线
下下一条指令		IF	ID	EX	…									：CPU 流水线
		IF	DF	EX	NA	SF								：FPU 流水线

• 运行说明 （单精度）

流水线中，CPU 流水线有 IF、ID、EX 的 3 个阶段，而 FPU 流水线有 IF、DF、E1、ED、ED、ED、ED、ED、ED、ED、E1、E2、SF 的 13 个阶段。即：进行 1 次 E1 阶段后，重复 7 次 ED 阶段，最后依次进入 E1、E2、 SF 阶段。

• 运行说明 （双精度）

流水线中，CPU 流水线有 IF、ID、EX 的 3 个阶段，而 FPU 流水线有 IF、DF、E1、E1、ED、ED、ED、ED、ED、ED、ED、ED、ED、ED、ED、ED、ED、ED、ED、ED、E1、E1、E1、E2、SF 的 26 个阶段。即：进行 2 次 E1 阶段后，重复 17 次 ED 阶段，最后依次进入 E1、 E1、 E1、 E2、 SF 阶段。

发生如 “8.6 由 FPU 引起的竞争”所示的竞争。在 FSQRT 流水线中 FSQRT 结果寄存器存取的指令发生重叠时，该指令等待到 FSQRT 指令执行结束。停止 E1 以后的阶段直到 FSQRT 执行结束，以后的指令也停止。因此，如果在从紧接着 FSQRT 指令之后到单精度时为 19 条、双精度时为 47 条指令以内，不配置 FSQRT 结果寄存器的浮点指令或者 FPU 相关的 CPU 指令，在此期间就可执行 CPU 指令或者其他 FPU 指令，提高其性能。

• 指令发行

该指令使用 FPU 算术运算流水线。关于竞争的详细内容请参照 “8.6 由 FPU 引起的竞争”。

该指令的 ED 阶段与槽无关，以状态运行。

• 并行执行可能性

无特别记载事项。

(18) 浮点比较指令

• 指令种类

```
FCMP/EQ     FRm,FRn
FCMP/GT     FRm,FRn
FCMP/EQ     DRm,DRn
FCMP/GT     DRm,DRn
```

• 流水线 （单精度）

	←→	←→	←→	←→	←→	←→	←→ 槽	
指令 A	IF	ID	EX					：CPU 流水线
	IF	DF	E1	E2				：FPU 流水线
下一条指令	IF	ID	EX	…				：CPU 流水线
	IF	DF	EX	NA	SF			：FPU 流水线
下下一条指令		IF	ID	EX	…			：CPU 流水线
		IF	DF	E1	E2	SF		：FPU 流水线

• 流水线 （双精度）

	←→	←→	←→	←→	←→	←→	←→	←→ 槽	
指令 A	IF	ID	EX	EX					：CPU 流水线
	IF	DF	E1	E1	E2				：FPU 流水线
下一条指令	IF	--	ID	EX	…				：CPU 流水线
	IF	--	DF	EX	NA	SF			：FPU 流水线
下下一条指令		IF	--	ID	EX	…			：CPU 流水线
		IF	--	DF	EX	NA	SF		：FPU 流水线

• 运行说明 （单精度）

流水线中，CPU 流水线有 IF、ID、EX 的 3 个阶段，而 FPU 流水线有 IF、DF、E1、E2 的 4 个阶段。因为 T 位通过 E2 确定，因此参考紧接之后的 T 位的指令停止 2 个周期。

FCMP	IF	ID	EX			：CPU 流水线
	IF	DF	E1	E2		：FPU 流水线
BT	IF	--	--	ID	EX	：CPU 流水线
	IF	--	--	DF	…	：FPU 流水线

• 运行说明 （双精度）

流水线中，CPU 流水线有 IF、ID、EX、EX 的 4 个阶段，而 FPU 流水线有 IF、DF、E1、E1、E2 的 5 个阶段。因为 T 位通过 E2 确定，因此参考紧接之后的 T 位的指令停止 3 个周期。

FCMP	IF	ID	EX				：CPU 流水线
	IF	DF	E1	E1	E2		：FPU 流水线
BT	IF	--	--	--	ID	EX	：CPU 流水线
	IF	--	--	--	DF	…	：FPU 流水线

• 指令发行

该指令使用 FPU 算术运算流水线。

• 并行执行可能性

双精度 FCMP 指令，不可与后续指令并行执行。

8.10 必要周期数的简单计算方法

以下说明必要周期数的简单计算方法。虽为粗略计算，但通过此方法可计算出执行目标指令列所必要的周期数。

此时，按照以下规则计算：

(1) 视指令已经取出，无需考虑取指令。
(2) 32位指令以执行状态周期运行。
(3) 发生资源竞争时，先发行的指令以执行状态周期操作。不可与后续指令并行执行。
(4) 紧接之后的指令使用先发行指令的结果时，算出先发行指令所需 “等待时间”周期。
(5) 紧接之后的指令不使用先发行指令的结果时，算出先发行指令所需的执行状态周期。
(6) 通过校正项简单地修正并行执行的效果。

以上情况还有很多例外，虽然精度达不到 100%, 但可计算出粗略的周期数。示例如下：

1. 通过数等待时间周期数进行计算时

		←→	←→	←→	←→	←→	←→	←→	←→	周期数
MOV.L	@R1, R0		ID	EX	MA	WB				2
ADD	#imm, R0		--	--	ID	EX				1
MOV.L	R0, @R1			--		ID	EX	MA		

因为 ADD 使用先行的 MOV.L 的结果，因此算出 MOV.L 执行所需等待时间周期数 （2 个周期）。又因为后续的 MOV.L 使用 ADD 的结果，因此算出 ADD 指令执行需要等待时间周期数 （1 个周期）。

2. 通过数执行状态数进行计算时

		←→	←→	←→	←→	←→	←→	←→	←→	周期数
MOV.L	@R1, R0		ID	EX	MA	WB				1→0
ADD	#imm, R2		ID	EX						1
MOV.L	R3, @R4			ID	EX	MA				

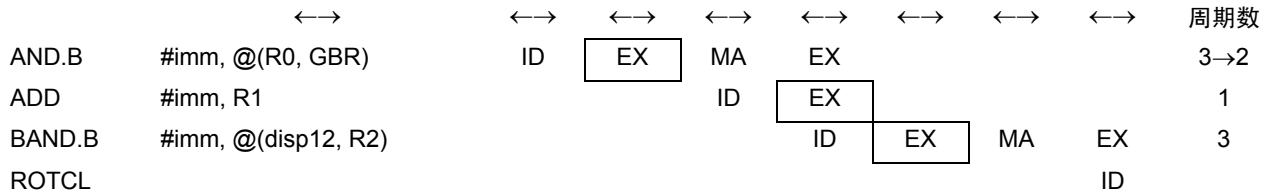
此时，由于后续指令不使用先发行的指令的结果，因此只要不发生资源竞争，各指令就并行执行。各指令执行所必要的周期数通过执行状态数，先行指令的执行状态数为 1 个时，可并行执行后续指令。并行执行时，算出先行指令执行所必要的周期数为执行状态数 –1，但是这作为校正简单地进行处理 （该校正作为后面所述公式的最终项出现）。

3. 有资源竞争时

		←→	←→	←→	←→	←→	←→	←→	←→	周期数
MOV.L	@R1, R0		ID	EX	MA	WB				1
MOV.L	@R3, R2			ID	EX	MA	WB			1
MOV.L	@R5, R4				ID	EX	MA	WB		

有资源竞争时，不可并行执行。各指令的执行必须要有 “执行状态”。

4. 执行状态数比1大的指令



将执行状态数比 1 大的指令视为此执行状态数渐渐减少，减少到 1 个时可与后续指令并行执行。此时，与后续指令并行执行时执行所必要的周期数为执行状态数 –1，未并行执行时为执行状态数。这里作为校正简单地进行处理 （该校正作为后面所述公式的最终项出现）。

以此为基础，简单考虑时，执行指令列所必要的周期数如下：存在有依存关系的区间与无依存关系的区间时，各自分别算出，结果相加。

- 各指令间全部都有依存 关系时。
  必要的周期数 = 全部指令的等待时间数相加后的周期数。

- 各指令间完全没有依存关系时。
  必要的周期＝全部指令的执行状态数相加后的周期数
  －（全部指令数－不可并行执行的指令数）÷2
  再次，所谓“不可并行执行的指令数”就是由于资源竞争而不可并行执行的指令数 （特别是紧接之前的指令为存储器存取指令、存储器存取指令）、执行状态数不为1的指令以及32位指令相加的数。
  根据该最终项，校正并行执行时的先行指令的必要周期的减少量。

例：各指令间全部都有依存关系时

```
BAND.B
 ROTCL
 BAND.B
 ROTCL
```

全部指令的 “等待时间”相加后为 8 个周期。

例：各指令间完全没有依存关系时

```
ADD       #imm, R0
BAND.B    #imm, @(disp12, R2)
MULR      R4, R0
ROTCL     R5
```

必要的周期数 =1+3+2+1–(4–2)÷2
=7–1=6 个周期

附录

附录 A. SH-2A/SH2A-FPU 并行执行性

- 根据所使用的运算器种类，通过下表判断可否并行执行。关于有多个分类的指令，确认各分类栏后，○时进行并行执行。

		第 2 指令										
		(1)BR	(2)MR	(3)MW	(4)MF	(5)ML	(6)MU	(7)SF	(8)FL	(9)FP	(10)FC	(11)EX
第 1 指令	(1)BR	×	○	○	○	○	○	○	○	○	○	○
	(2)MR	○	×	×	○	○	○	○	○	○	○	○
	(3)MW	○	×	×	×	○	○	○	○	○	○	○
	(4)MF	○	○	×	×	○	○	○	○	○	○	○
	(5)ML	○	○	○	○	×	○	○	○	○	○	○
	(6)MU	○	○	○	○	○	×	○	○	○	○	○
	(7)SF	○	○	○	○	○	○	×	○	○	○	○
	(8)FL	○	○	○	○	○	○	○	×	○	×	○
	(9)FP	○	○	○	○	○	○	○	○	×	×	○
	(10)FC	×	×	×	×	×	×	×	×	×	×	×
	(11)EX	○	○	○	○	○	○	○	○	○	○	○

第 1 指令时的分类	第 2 指令时的分类	指令					
BR	BR	BF	disp	BF/S	disp	BT	disp
		BT/S	disp	BSR	disp	BSRF	Rm
		BRA	disp	BRAF	Rm	JMP	@Rm
		JSR	@Rm	JSR/N	@Rm	RTS	
		RTS/N		RTV/N	Rm	TRAPA	#imm
MR	MR	LDC.L	@Rm+,GBR	LDC.L	@Rm+,VBR	LDS.L	@Rm+,PR
		MOV.B	@(disp,GBR),R0	MOV.B	@(disp,Rm),R0	MOV.B	@(R0,Rm),Rn
		MOV.B	@Rm,Rn	MOV.B	@Rm+,Rn	MOV.B	@-Rm,R0
		MOV.B	@(disp12,Rm),Rn	MOV.W	@(disp,GBR),R0	MOV.W	@(disp,Rm),R0
		MOV.W	@(R0,Rm),Rn	MOV.W	@Rm,Rn	MOV.W	@Rm+,Rn
		MOV.W	@-Rm,R0	MOV.W	@(disp12,Rm),Rn	MOV.W	@(disp,PC),Rn
		MOV.L	@(disp,GBR),R0	MOV.L	@(disp,Rm),Rn	MOV.L	@(R0,Rm),Rn
		MOV.L	@Rm,Rn	MOV.L	@Rm+,Rn	MOV.L	@-Rm,R0
		MOV.L	@(disp12,Rm),Rn	MOV.L	@(disp,PC),Rn	MOVU.B	@(disp12,Rm),Rn
		MOVU.W	@(disp12,Rm),Rn	MOVML.L	@R15+,Rn	MOVMU.L	@R15+,Rn
		PREF	@Rn				

第 1 指令时的分类	第 2 指令时的分类	指令		
MW	MR	AND.B #imm,@(R0,GBR)	BCLR.B #imm3,@(disp12,Rn)	BSET.B #imm3,@(disp12,Rn)
		BST.B #imm3,@(disp12,Rn)	OR.B #imm,@(R0,GBR)	STC.L SR,@-Rn
		TAS.B @Rn	XOR.B #imm,@(R0,GBR)	
MW	MW	MOV.B R0,@(disp,GBR)	MOV.B R0,@(disp,Rn)	MOV.B Rm,@(R0,Rn)
		MOV.B Rm,@Rn	MOV.B Rm,@-Rn	MOV.B R0,@Rn+
		MOV.B Rm,@(disp12,Rn)	MOV.W R0,@(disp,GBR)	MOV.W R0,@(disp,Rn)
		MOV.W Rm,@(R0,Rn)	MOV.W Rm,@Rn	MOV.W Rm,@-Rn
		MOV.W R0,@Rn+	MOV.W Rm,@(disp12,Rn)	MOV.L R0,@(disp,GBR)
		MOV.L Rm,@(disp,Rn)	MOV.L Rm,@(R0,Rn)	MOV.L Rm,@Rn
		MOV.L Rm,@-Rn	MOV.L R0,@Rn+	MOV.L Rm,@(disp12,Rn)
		MOVML.L Rm,@-R15	MOVMU.L Rm,@-R15	STC.L GBR,@-Rn
		STC.L VBR,@-Rn	STS.L PR,@-Rn	
ML	ML	STS MACH,Rn	STS MACL,Rn	
MU	MU	CLRMAC	DMULS.L Rm,Rn	DMULU.L Rm,Rn
		MUL.L Rm,Rn	MULS.W Rm,Rn	MULU.W Rm,Rn
		LDS Rm,MACL	LDS Rm,MACH	
ML,MU	ML	MULR R0,Rn		
SF	SF	DIVU R0,Rn	EXTS.B Rm,Rn	EXTS.W Rm,Rn
		EXTU.B Rm,Rn	EXTU.W Rm,Rn	ROTCL Rn
		ROTCR Rn	ROTL Rn	ROTR Rn
		SHAD Rm,Rn	SHAL Rn	SHAR Rn
		SHLD Rm,Rn	SHLL Rn	SHLL16 Rn
		SHLL2 Rn	SHLL8 Rn	SHLR Rn
		SHLR16 Rn	SHLR2 Rn	SHLR8 Rn
		SWAP.B Rm,Rn	SWAP.W Rm,Rn	XTRCT Rm,Rn
FL	FL	FABS DRn	FABS FRn	FLDI0 FRn
		FLDI1 FRn	FLDS FRm,FPUL	FMOV DRm,DRn
		FMOV FRm,FRn	FNEG DRn	FNEG FRn
		FSTS FPUL,FRn		
ML,FL	ML,FL	STS FPUL,Rn		
FP	FP	FADD DRm,DRn	FADD FRm,FRn	FCMP/EQ FRm,FRn
		FCMP/GT FRm,FRn	FCNVDS DRm,FPUL	FCNVSD FPUL,DRn
		FDIV DRm,DRn	FDIV FRm,FRn	FLOAT FPUL,DRn
		FLOAT FPUL,FRn	FMAC FR0,FRm,FRn	FMUL DRm,DRn
		FMUL FRm,FRn	FSCHG	FSQRT DRn
		FSQRT FRn	FSUB DRm,DRn	FSUB FRm,FRn
		FTRC DRm,FPUL	FTRC FRm,FPUL	
FC	FC	FCMP/EQ DRm,DRn	FCMP/GT DRm,DRn	
ML,FC	ML,FC	STS FPSCR,Rn		

第 1 指令时的分类	第 2 指令时的分类	指令		
EX	EX	ADD #imm,Rn	ADD Rm,Rn	ADDC Rm,Rn
		ADDV Rm,Rn	AND #imm,R0	AND Rm,Rn
		BCLR #imm3,Rn	BLD #imm3,Rn	BSET #imm3,Rn
		BST #imm3,Rn	CLRT	CMP/EQ #imm,R0
		CMP/EQ Rm,Rn	CMP/GE Rm,Rn	CMP/GT Rm,Rn
		CMP/HI Rm,Rn	CMP/HS Rm,Rn	CMP/PL Rn
		CMP/PZ Rn	CMP/STR Rm,Rn	CLIPS.B Rn
		CLIPS.W Rn	CLIPU.B Rn	CLIPU.W Rn
		DIV0S Rm,Rn	DIV0U	DIVS R0,Rn
		DIV1 Rm,Rn	DT Rn	LDC Rm,GBR
		LDC Rm,SR	LDC Rm,TBR	LDC Rm,VBR
		LDS Rm,PR	LDBANK @Rm,R0	MOV #imm,Rn
		MOV Rm,Rn	MOVA @(disp,PC),R0	MOVI20 #imm20,Rn
		MOVI20S #imm20,Rn	MOVT Rn	MOVRT Rn
		NEG Rm,Rn	NEGC Rm,Rn	NOP
		NOT Rm,Rn	NOTT	OR #imm,R0
		OR Rm,Rn	SETT	STC GBR,Rn
		STC SR,Rn	STC TBR,Rn	STC VBR,Rn
		STS PR,Rn	STBANK R0,@Rn	SUB Rm,Rn
		SUBC Rm,Rn	SUBV Rm,Rn	TST #imm,R0
		TST Rm,Rn	XOR #imm,R0	XOR Rm,Rn
		RESBANK(BO==0)		
MR,MU	MR,MU	LDS.L @Rm+,MACH	LDS.L @Rm+,MACL	
MW,ML	MW,ML	STS.L MACH,@-Rn	STS.L MACL,@-Rn	
MW,FL	MW,FL	FMOV.S @(R0,Rm),FRn	FMOV.S @Rm,FRn	FMOV.S @Rm+,FRn
		FMOV.S @(disp12,Rm),FRn	FMOV.S FRm,@(R0,Rn)	FMOV.S FRm,@-Rn
		FMOV.S FRm,@Rn	FMOV.S FRm,@(disp12,Rn)	FMOV.D @(R0,Rm),DRn
		FMOV.D @Rm,DRn	FMOV.D @Rm+,DRn	FMOV.D @(disp12,Rm),DRn
		FMOV.D DRm,@(R0,Rn)	FMOV.D DRm,@-Rn	FMOV.D DRm,@Rn
		FMOV.D DRm,@(disp12,Rn)		
MF,FL	MF,FL	LDS Rm,FPUL		
MF,FC	MF,FC	LDS Rm,FPSCR		
MR,FC	MR,FC	LDS.L @Rm+,FPSCR	LDS.L @Rm+,FPUL	
MW,ML,FC	MW,ML,FC	STS.L FPSCR,@-Rn	STS.L FPUL,@-Rn	
BR	MR	JSR/N @@(disp8,TBR)		
MR,MU	MR	RESBANK(BO==1)		
EX	MR	BAND.B #imm3,@(disp12,Rn)	BANDNOT.B #imm3,@(disp12,Rn)	BLD.B #imm3,@(disp12,Rn)
		BLDNOT.B #imm3,@(disp12,Rn)	BOR.B #imm3,@(disp12,Rn)	BORNOT.B #imm3,@(disp12,Rn)
		BXOR.B #imm3,@(disp12,Rn)	LDC.L @Rm+,SR	RTE
		SLEEP	TST.B #imm,@(R0,GBR)	
MU	MR	MAC.W @Rm+,@Rn+	MAC.L @Rm+,@Rn+	

- 多步指令在第一步和最后步进行并行执行。
- FPU指令沿用SH4的分类 （①LS类型②FE类型③CO类型）。新指令的32位FMOV指令分类为①LS类型。
- 原则上如果之前的指令为多阶段指令则32位指令可并行执行。不可与之后的指令并行执行。但是，存储器-Tbit间位操作指令相互的组合可并行执行。
- MOVMU.L,MOVML.L指令不可与之前的指令并行执行。
- 延迟转移指令与延迟槽不可并行执行。

- 多步指令：
  TRAPA, MOVMU.L, MOVML.L, AND.B, OR.B, TST.B, XOR.B, TAS.B, BCLR.B, BSET.B, BST.B, BAND.B, BANDNOT.B, BLD.B, BLDNOT.B, BOR.B, BORNOT.B, BXOR.B, MUL.L, DMULS.L, DMULU.L, MULR, DIVU, DIVS, FCMP/EQ DRm,DRn, FCMP/GT DRm,DRn, LDC Rm,SR, STC SR,Rn, LDC.L @Rm+,SR, STC.L SR,@-Rn, LDBANK, STBANK, RESBANK, FMOV.D, FMOV DRm,DRn, JSR/N @@(disp,TBR), SLEEP, RTE, MAC.W, MAC.L
- 32位指令：
  MOVI20, MOVI20S, MOV.B @(disp12,Rm),Rn, MOV.W @(disp12,Rm),Rn, MOV.L @(disp12,Rm),Rn, MOV.B Rm,@(disp12,Rn), MOV.W Rm,@(disp12,Rn), MOV.L Rm,@(disp12,Rn),MOVU.B, MOVU.W, FMOV.S @(disp12,Rm),FRn, FMOV.D @(disp12,Rm),DRn, FMOV.S FRm,@(disp12,Rn), FMOV.D DRm,@(disp12,Rn), BCLR.B, BSET.B, BST.B, BAND.B, BANDNOT.B, BLD.B, BLDNOT.B, BOR.B, BORNOT.B, BXOR.B
- 32位FMOV指令：
  FMOV.S @(disp12,Rm),FRn, FMOV.D @(disp12,Rm),DRn, FMOV.S FRm,@(disp12,Rn), FMOV.D DRm,@(disp12,Rn),
- 存储器－Tbit间位操作指令：
  BAND.B, BANDNOT.B, BLD.B, BLDNOT.B, BOR.B, BORNOT.B, BXOR.B
- 延迟转移指令：
  BRA, BSR, BRAF, BSRF, JMP, JSR, RTS, RTE, BT/S, BF/S

附录 B. 编程的准则 （关于 MOVI20、 MOVI20S 的使用方法）

SH2A、 SH2A-FPU 中， MOVI20#imm20,Rn 及 MOVI20S #imm20,Rn 指令减少因 PC 相对指令产生的文字存取，具有提高周期性能的效果。为了获得这些指令的效果，推荐汇编程序中如下表述：

(1) MOVI20 的使用方法

MOVI20S 进行符号扩展。根据这个指令，能使用 H'00000000 ～ H'0007FFFF， H'FFF80000 ～ H'FFFFFFFF 的范围。

连续配置以下的指令列。

```
MOVI20 #imm20, Rn
无条件转移指令 *1
```

例 .

```
MOVI20 #imm20, Rn
JMP @Rm
```

(2) MOVI20S 的使用方法

MOVI20S 进行符号扩展。并通过与 ADD#imm, Rn 指令组合，能使用 H'00000000 ～ H'07FFFF7F， H'F7FFFF80 ～ H'FFFFFFFF 的范围。

连续配置以下的指令列。

```
MOVI20S #imm20, Rn
ADD #imm, Rn
无条件转移指令 *1
```

例 .

```
MOVI20S #imm20, Rn
ADD #imm, Rn
JMP @Rm
```

【注】 1. H'07FF FF80 ～ H'07FF FFFF 范围的指定为

```
MOVI20S #imm20, R0
OR #imm, R0
无条件转移指令 *1
```

或者通过如下所示的 32 位地址读操作来进行指定。

```
MOV.L @(disp, PC), Rn
无条件转移指令 *1
```

*1 无条件转移指令：BRAF Rm, BSRF Rm, JMP @Rm, JSR @Rm, JSR/N @Rm

修订记录	SH-2A、 SH2A-FPU 软件手册

Rev.	发行日	修订内容	
		页	修订处
1.00	2008.12.17	—	初版发行

瑞萨 32 位 RISC 单片机
软件手册
SH-2A、 SH2A-FPU

Publication Date: Rev1.00, Dec.17, 2008
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-2A、 SH2A-FPU

瑞萨电子株式会社

RCJ09B0061-0100