To our customers,

Old Company Name in Catalogs and Other Documents

On April 1st, 2010, NEC Electronics Corporation merged with Renesas Technology Corporation, and Renesas Electronics Corporation took over all the business of both companies. Therefore, although the old company name remains in this document, it is a valid Renesas Electronics document. We appreciate your understanding.

Renesas Electronics website: http://www.renesas.com

April 1st, 2010
Renesas Electronics Corporation

Issued by: Renesas Electronics Corporation (http://www.renesas.com)

Send any inquiries to http://www.renesas.com/inquiry.

RENESAS

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.

RENESAS APPLICATION NOTE

SuperH RISC engine C/C++ Compiler Package

APPLICATION NOTE: [Compiler use guide] SH-2A / SH2A-FPU

This document explains the coding techniques for SH-2A / SH2A-FPU, as well as how to use C extended functions and compile options, for the SuperH RISC engine C/C++ Compiler V.9.01.

1. Overview of SH-2A / SH2A-FPU

SH-2A / SH2A-FPU (herein as *SH-2A*) is a new SH microcomputer compatible on the object code level with SH-1 and SH-2. SH-2A offers the following features:

- Newly added instructions, for improved calculation processing performance and reduced program size
- A register bank, for reduced interrupt response times
- A jump table base register (TBR), for faster subroutine calls
- A two-way superscalar, for concurrent execution of two instruction
- A new Harvard architecture

This document explains how to use these features with the SuperH RISC engine C/C++ compiler.

2. Features and Usage for New SH-2A Instructions

Table 1.1 lists the features of the new instructions for SH-2A.

Table 1.1 Features of the new instructions for SH-2A

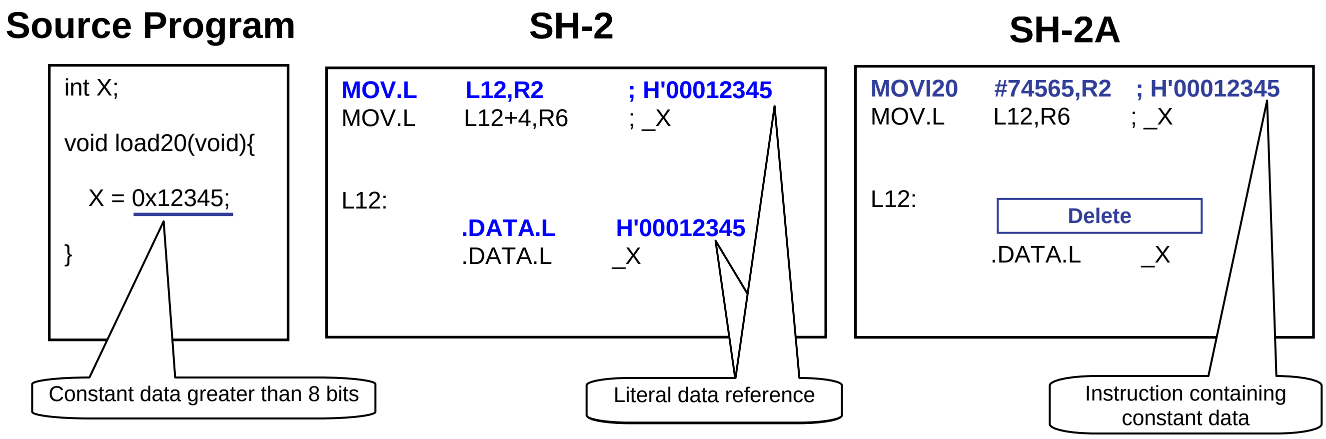
Item	Contents	Effects	
		Speed	Size
20-bit long immediate load (32-bit fixed-length instruction)	An instruction that transfers 20-bit immediate data within an instruction code to a register, effective for loading addresses and constants.	Up Memory access reduction	Small
Relative load/store for 12-bit registers with `disp` (32-bit fixed-length instruction)	An instruction for referencing memory by specifying 12-bit displacement, improving access efficiency for structures.	Up	Small
Load multi / Store multi	An instruction that saves and restores multiple consecutive registers to memory in one instruction, reducing the code size for register save/restore processing.	Even	Small
Auto-increment/decrement	Performs auto-increment/decrement for pointers, as used in consecutive array access.	Up	Small
Division instruction	An instruction for performing 32-bit / 32-bit division, allowing run-time routine calls to be eliminated.	Up	Small
Multiplication instruction	An instruction for performing 32-bit x 32-bit multiplication, storing the lower 32 bits of the calculation results in the general register Rn, to reduce access to the MAC register.	Up	Small
Saturation value comparison instruction	An instruction that performs comparison with a saturation value, returning the maximum saturation when the value is higher, or the maximum saturation when the value is lower. Significantly improves efficiency in determining processing for overflows and underflows.	Up	Small
Bitwise operation instructions (32-bit fixed-length instruction)	Performs 1-bit operations in memory, including logic calculations, operations, acquisition, inverted acquisition, and inverted logic calculations).	Up	Small
Branching instruction without delay slots	A branching instruction without delay slots, for deleting unnecessary NOP instructions to reduce code size.	Even	Small
Barrel shift instruction	Instructions for shifting arbitrary bits, including arithmetic shifting and logical shifting.	Up	Small

2.1 20-bit Long Immediate Load

Format: `MOVI20 #imm20,Rn`, `MOVI20S #imm20,Rn`

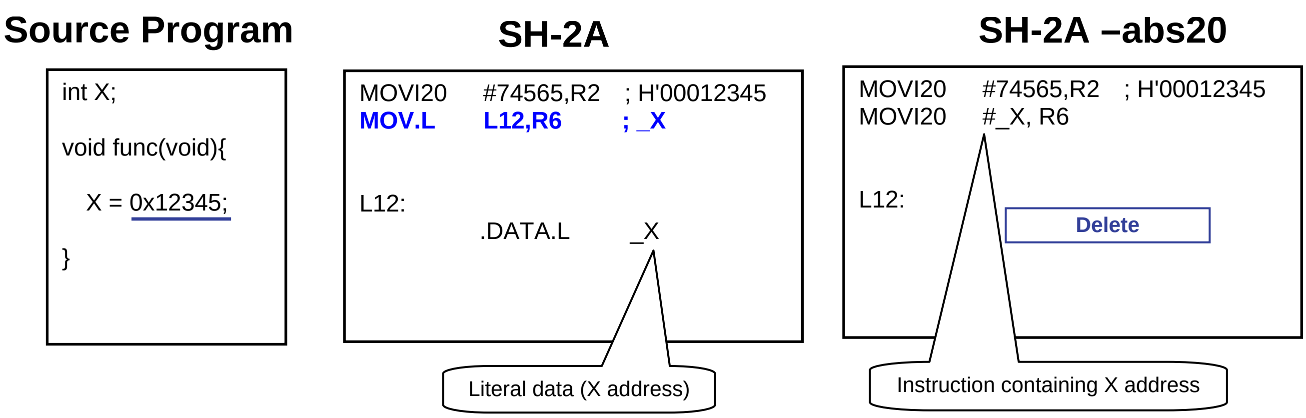
This instruction transfers 20-bit immediate data, such as addresses for constants and functions. For example, when a constant longer than 8 bits is handled in SH-2, literal data needs to be used. Since SH-2A allows immediate data of up to 20 bits to be embedded in an instruction, both code size and memory access can be reduced. Note that this instruction is 32 bits long.

Example 2-1(1)



Since the addresses of functions and variables are also 32 bits, they are used with literal data. In Example 2-1(1), the address for external variable *X* is used with literal data for both SH-2 and SH-2A. But when addresses for functions and variables are expressed as 20 bits, `#pragma abs20` or the `-abs20` compile option can be specified to generate code in which 20-bit immediate load instructions are used.

Example 2-1(2)



Using `#pragma abs20`

When `#pragma abs20` *id* [,…] is specified before a variable or function is declared or defined, the address of the specified variable or function is expressed in 20 bits. `#pragma abs20` needs to be specified in all source code in which the corresponding variable or function is referenced. We recommend that you specify #pragma asb20 in header file included for all source files.

Example 2-1(3)

Variable X and function func1 specified for abs20

common.h

```
#pragma abs20(X, func1)

extern int X;
extern void func1(void);

extern void func2(void);
```

source1.c

```
#include "common.h"

int X;

void func1(void){
      X = 1;
}
```

source2.c

```
#include "common.h"

void func2(void){
      func1();
}
```

About `-abs20`

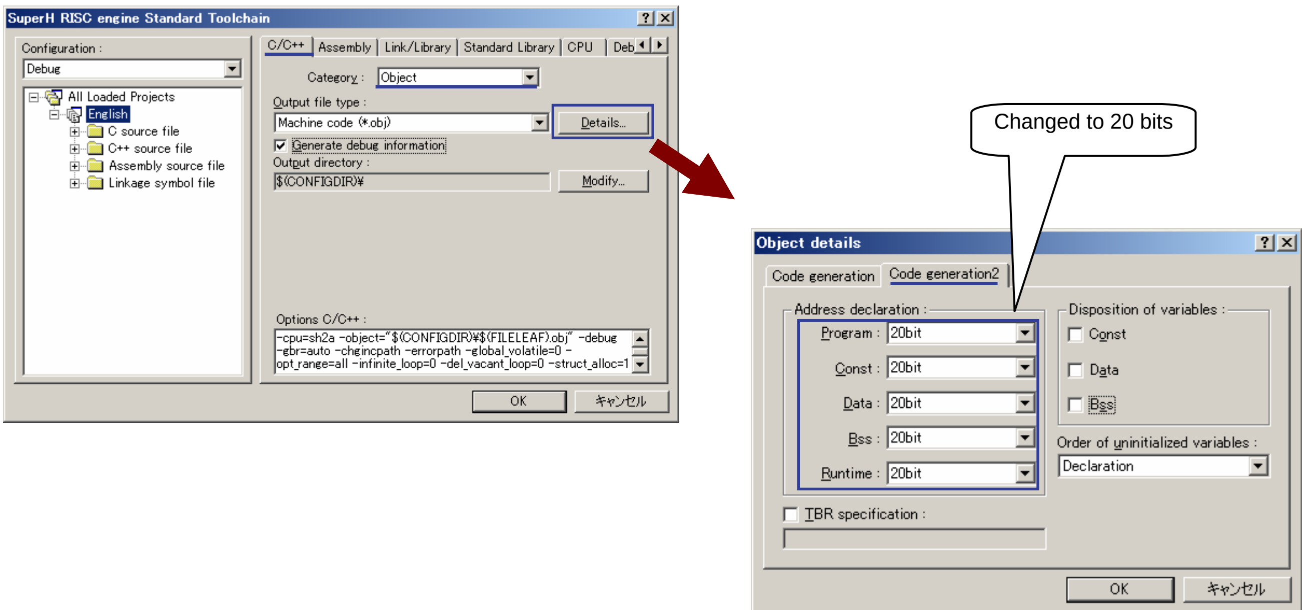
The `-abs20` compile option can be specified to change the address expressions for all functions and variables in the corresponding source file to 20-bit expressions. The `-abs20` option has the following subcommands:

Format: `-abs20` = { `Program` | `Const` | `Data` | `Bss` | `Run` | `All` }

- `Program`: Targets the program area
- `Const`: Targets the constants area
- `Data`: Targets the initialized data area
- `Bss`: Targets the uninitialized data area
- `Run`: Targets run-time routines
- `All`: Targets all areas

Specifying `-abs20` in High-performance Embedded Workshop (herein as *HEW*)

`-abs20` can be specified in the Tool Chain dialog box.



Precautions regarding `#pragma abs20` and `-abs20` usage

Functions and variables for which `#pragma abs20` or `-abs20` is specified must be placed within the following address space range.

Keep in mind that the program may malfunction if the function or variable is outside of this range.

Table 1-1 Range for 20-bit address expressions

Address range	
Minimum	Maximum
0x00000000	0x0007FFFF
0xFFF80000	0xFFFFFFFF

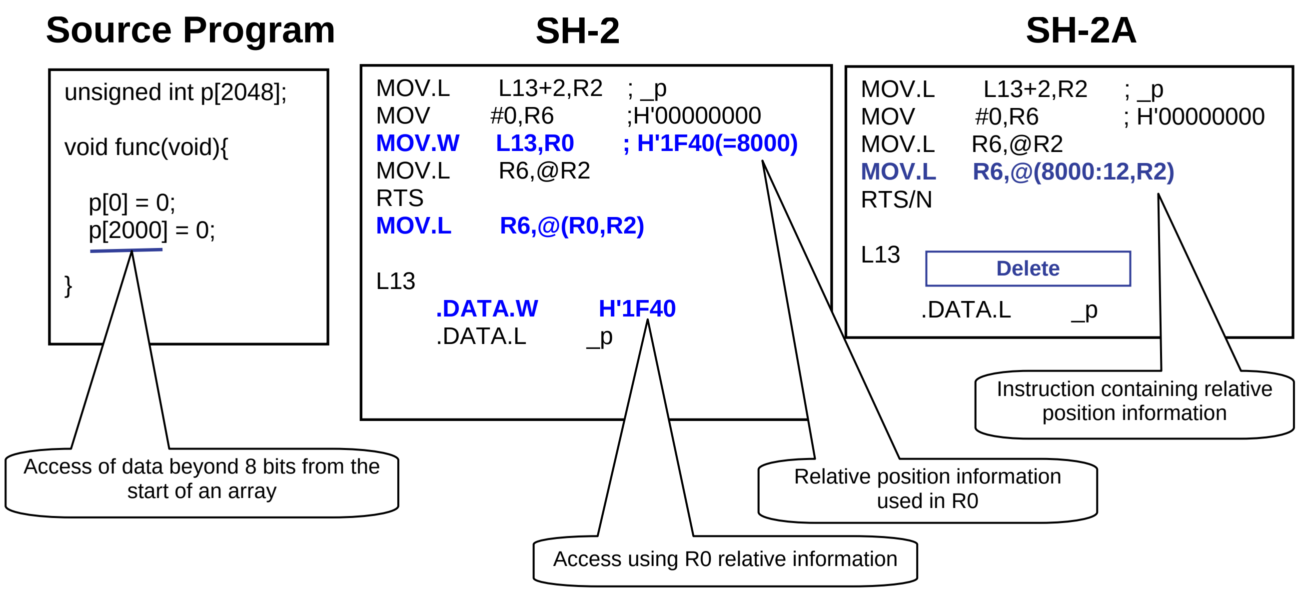
Note that if both `-abs20` and `#pragma abs16|abs20|abs28|abs32` are specified, the latter takes effect.

2.2 Relative Load/Store for 12-bit Registers with Disp

Format: `MOV.B Rm,@(disp 12,Rn)`, `MOV.W Rm,@(disp 12,Rn)`, `MOV.L Rm,@(disp 12,Rn)`
`MOV.B @(disp 12,Rn),Rm`, `MOV.W @(disp 12,Rn),Rm`, `MOV.L @(disp 12,Rn),Rm`

This is a transfer instruction with a 12-bit `disp`, and can be used for data access within arrays, structures, and stacks. Since SH-2 only allows relative access instructions for registers with 8-bit `disp`, when array or other data placed more than 8 bits out is accessed (which happens frequently), code in which an offset value is used as literal data is generated. Since SH-2A allows offset values of up to 12 bits to be embedded in instructions, it can access data in a larger range more efficiently than SH-2. Note that this instruction is 32 bits long.

Example 2-2



When using this instruction with a structure or array, make sure that it can fit within a size for which 12-bit `disp` can be used (4,095 elements or fewer for an array). Note that no special `#pragma` specification is needed when this instruction is used.

2.3 Load Multi / Store Multi

Format: `MOVMUL.L Rm,@-R15`, `MOVMUL.L @R15+,Rn`
`MOVMU.L Rm,@-R15`, `MOVMU.L @R15+,Rn`

This instruction loads or stores multiple registers, in one instruction. The number of instruction execution states is not different than that when each register is specified individually, but instruction size can be reduced because everything can be specified in one instruction. This instruction can be used for saving or restoring registers during function calls. The compiler uses this instruction automatically, and there are no particular items of concern about during coding.

Example 2-3

Source Program

```
void func(void){
   ..........
}
```

SH-2

```
_func:
MOV.L    R8,@-R15
MOV.L    R9,@-R15
MOV.L    R10,@-R15
MOV.L    R11,@-R15
MOV.L    R12,@-R15
MOV.L    R13,@-R15
MOV.L    R14,@-R15
STS.L    PR,@-R15
       ...........
```

```
LDS.L    @R15+,PR
MOV.L    @R15+,R14
MOV.L    @R15+,R13
MOV.L    @R15+,R12
MOV.L    @R15+,R11
MOV.L    @R15+,R10
MOV.L    @R15+,R9
RTS
MOV.L    @R15+,R8
```

SH-2A

```
_func:
MOVMU.L    R8,@-R15
       ..........
MOVMU.L    R8,@-R15
RTS/N
```

2.4 Auto-increment / decrement

Format: (1) `MOV.B R0,@Rn+`, `MOV.W R0,@Rn+`, `MOV.L R0,@Rn+`
(2) `MOV.B @-Rn,R0`, `MOV.W @-Rn,R0`, `MOV.L @-Rn,R0`

(1) The `R0` register is transferred to the address indicated by the `Rn` register, and the value of the `Rn` register is incremented by 1 for `.B`, 2 for `.W`, and 4 for `.L`.
(2) The data in the address indicated by the `Rm` register is transferred to the `R0` register, and the value of the `Rm` register is decremented by 1 for `.B`, 2 for `.W`, and 4 for `.L`.

This instruction can be used to access array elements sequentially. The compiler uses this instruction automatically, and there are no particular items of concern about during coding.

Example 2-4

Source Program

```
int array[20];

void gunc(int s[20]){
   int I;

   for(i=0, i<20, i++){
      array[i] = s[i];
   }
}
```

SH-2

```
_func:
   MOV     #20,R5       ; H'00000014
   MOV.L   L17,R6       ; _aray

L15:
   MOV.L   @R4+,R2
   DT      R5
   MOV.L   R2,@R6
   BF/S    L15
   ADD     #4,R6
   RTS
   NOP

L17
   .DATA.L      _aray
```

SH-2A -abs20

```
_func:
   MOV       #20,R5     ; H'00000014
   MOVI20    #_aray,R6

L14:
   MOV.L   @R4+,R0
   DT      R5
   BF/S    L14
   MOV.L   R0,@R6+
   RTS/N
```

2.5 Division Instruction

Format: `DIVS R0,Rn` (36 execution states), `DIVU R0,Rn` (34 execution states)

This instruction divides the 32-bit contents of the `Rn` register (dividend) by the contents of the `R0` register (divisor). When 32-bit division is performed on SH-2, the result is calculated by calling a 1-bit division instruction multiple times. The compiler provides a run-time routine (runtime library) to perform division processing by calling this 1-bit division instruction multiple times. Since SH-2A supports 32-bit division instructions, it performs division processing by instruction, instead of run-time routine, allowing improved execution speed and reduced program size.

Example 2-5(1)

Source Program

```
int A;

void func(int d){

   A  =  A / d;

}
```

SH-2

```
_func:
   STS.L     PR,@-R15
   MOV.L     L18,R6    ; _A
   MOV.L     L18+4,R2  ; __divls
   MOV.L     @R6,R1
   JSR       @R2
   MOV       R4,R0
   LDS.L     @R15+,PR
   RTS
   MOV.L     R0,@R6

L18
   .DATA.L       _A
   .DATA.L       __divls
```

Run-time routine call

SH-2A -abs20

```
_func:
   MOVI20    #_A,R6
   MOV.L     @R6,R2
   MOV       R4,R0
   DIVS      R0,R2
   RTS
   MOV.L     R2,@R6
```

Division by integer constants

When the divisor is an integer constant, an optimization can be performed in which division is processed as multiplication. SH-2A supports division instruction, but since the number of execution states for division instructions is 34 or 36, processing using multiplication can be performed faster. Note that since there are more instructions, the program size increases. Optimization in which processing is performed by multiple by the inverse divisor can be specified by the `DIvision=cpu= Inline |Runtime` compile option. Note that the default setting is different for speed and size optimizations. When speed first or size & speed is specified, this optimization is applied by default. When size first is specified, this optimization is not applied by default (see *Table 2-5*). Note that for SHC compiler V.9.00, optimization in which constant division is performed as multiplication cannot be used. In this case, calculation is always performed using division instructions.

Table 2-5 Calculation method for constant division

CPU	SH-2	SH-2A (SHC V.9.00)	SH-2A (SHC V.9.01)
Size first (`-size`)	Runtime	Runtime	Runtime
Size & speed	Inline	Runtime	Inline
Speed first (`-speed`)	Inline	Runtime	Inline

SH-2

Inline: Calculation in which constant division is converted to multiplication is performed. A run-time routine is called for variable division.

Runtime: A run-time routine is called for constant division and variable division for which no shift calculation is performed.

SH-2A

Inline: Calculation in which constant division is converted to multiplication is performed. Variable division uses division instructions.

Runtime: Division instructions are used for constant division and variable division for which no shift calculation is performed.

Example 2-5(2)

Source Program	SH-2A -size -abs20	SH-2A -speed -abs20 -macsave=0

```
unsigned int A;

void func(void){

   A   =   A / 10;

}
```

```
_func:
      MOVI20    #_A,R5
      MOV.L     @R5,R6
      MOV       #10,R0       ;H'0000000A
      DIVU      R0,R6
      RTS
      MOV.L     R6,@R5
```

```
_func:
   MOVI20     #_A,R6
   MOV.L      L12+4,R1    ; H'CCCCCCCD
   MOV.L      @R5,R6      ; A
   DMULU.L    R6,R1
   STS        MACH,R2
   MOV.L      R2,@R6
   SHLR2      R2
   SHLR       R2
   RTS
   MOV.L      R2,@R5      ; A

L12:
   .DATA.L    H'CCCCCCCD
```

2.6 Multiplication Instruction

Format: `MULR  R0,Rn`

This instruction performs 32-bit multiplication of the contents of the `R0` general register with `Rn`, and stores the bottom 32 bits of the results in the `Rn` general register. Since the multiplication instruction supported by SH-2 (`MUL.L Rm,Rn`) stores the calculation results in the MACL register, code is required to copy the results from MACL to the general register. Using MULR instructions makes such copy processing unnecessary, and can improve performance in both speed and code size. This instruction is used automatically by the compiler, and there are no particular items of concern about during coding.

Example 2-6

Source Program	SH-2	SH-2A -abs20

```
int A;

void func(int m){

   A   =   A / m;

}
```

```
_func:
   STS.L      MACL,@-R15
   MOV.L      L18,R6    ; A
   MOV.L      @R6,R1
   MUL.L      R1,R4
   STS        MACL,R5
   MOV        R5,@R6
   RTS
   LDS.L      @R15+,MACL

L18
   .DATA.L       _A
```

```
_func:
   MOVI20     #_A,R5
   MOV.L      @R5,R6
   MOV        R6,R0
   MULR       R0,R6
   RTS
   MOV.L      R6,@R5
```

2.7 Saturation Value Comparison Instruction

Format: `CLIPS.B Rn`, `CLIPS.W Rn`, `CLIPU.B Rn`, `CLIPU.W Rn`

This instruction determines saturation. When the Rn general register is more than the maximum saturation, the maximum saturation is stored in Rn. When the Rn general register is less than the minimum saturation, the minimum saturation is stored in Rn. In both cases, the CS bit is set to 1. This instruction can be used to make saturation determination processing significantly faster. Use the embedded function to use this instruction.

Embedded function	Description	Minimum	Maximum
`long clipsb (long data)`	When the data is between -128 and 127, the value is returned. When the data is out of range, the maximum or minimum is returned.	-128	127
`long clipsw (long data)`	When the data is between -32768 and 32767, the value is returned. When the data is out of range, the maximum or minimum is returned.	-32768	32767
`unsigned long clipub (unsigned long data)`	When the data is between 0 and 255, the value is returned. When the data is out of range, the maximum or minimum is returned.	0	255
`unsigned long clipuw (unsigned long data)`	When the data is between 0 and 65535, the value is returned. When the data is out of range, the maximum or minimum is returned.	0	65535

Example 2-7

SH-2

C source code

```
unsigned long result,x,y;

void func(void){

    result = x * y;
    if( result >255)
        result = 255;
}
```

Generated code

```
    STS.L       MACL,@-R15
    MOV.L       L23+44,R1
    MOV         #1,R6
    MOV.L       @R1,R2
    MOV.L       L23+48,R1
    MOV.L       L23+52,R5
    MOV.L       @R1,R4
    SHLL8       R6
    MUL.L       R2,R4
    STS         MACL,R7
    CMP/HI      R6,R7
    BF/S        L19
    MOV.L       R7,@R5
    MOV         #-1,R2
    EXTU.B      R2,R2
    MOV.L       R2,@R5
L19:
    RTS
    LDS.L       @R15+,MACL
```

SH-2A (Embeded function usage)

C source code

```
#include <machine.h>
unsigned long result,x,y;

void func(void){

    result = clipub( x * y );
}
```

Generated code

```
    MOVI20      #_x,R1
    MOVI20      #_y,R4
    MOV.L       @R1,R7
    MOV.L       @R4,R0
    MOV.L       L19+16,R2   ; _result
    MULR        R0,R7
    CLIPU.B     R7
    RTS
    MOV.L       R7,@R2
```

2.8 Bitwise Operation Instructions

Format:

Bitwise AND:	`BAND.B #imm3,@(disp12,Rn)` (3 execution states)
Bitwise not AND:	`BANDNOT.B #imm3,@(disp12,Rn)` (3 execution states)
Bitwise clear:	`BCLR.B #imm3,@(disp12,Rn)` (3 execution states)
16-bit instruction:	`BCLR #imm3,Rn` (1 execution state)
Bitwise load:	`BLD.B #imm3,@(disp12,Rn)` (3 execution states)
16-bit instruction:	`BLD #imm3,Rn` (3 execution states)
Bitwise not load:	`BLDNOT.B #imm3,@(disp12,Rn)` (3 execution states)
Bitwise OR:	`BOR.B #imm3,@(disp12,Rn)` (3 execution states)
Bitwise not OR:	`BORNOT.B #imm3,@(disp12,Rn)` (3 execution states)
Bitwise set:	`BSET.B #imm3,@(disp12,Rn)` (3 execution states)
16-bit instruction:	`BSET #imm3,Rn` (1 execution state)
Bitwise store:	`BST.B #imm3,@(disp12,Rn)` (3 execution states)
16-bit instruction:	`BST #imm3,Rn` (1 execution state)
Bitwise XOR:	`XBOR.B #imm3,@(disp12,Rn)` (3 execution states)

Bitwise operation calculations are executed in one instruction. Each calculation is performed for the bit specified by `imm3` in memory at the address indicated by (disp12+Rn). These instructions are 32 bits long (though some are 16 bits).

SH-2 performs the following three instructions when bitwise operations are performed:

1. It reads the memory value of the target address area.
2. It gets the bitwise operated value of the value read from memory using an AND instruction or OR instruction.
3. It writes the memory value obtained by calculation to the target address area.

If an interrupt occurs while these three instructions are being performed, and the value of a target address area for the bitwise operation changes during interrupt processing, the results of the bitwise operation processing will be invalid. As such, when bitwise operations are performed on address areas that might be overwritten within the interrupt function, interrupts need to be prohibited before processing occurs. Note that since bitwise operations can be performed in one operation when the bitwise operation instructions added to SH-2A are used, interrupts no longer need to be prohibited, improving both processing speed and interrupt responsiveness.

Since these instructions are generated by optimization, keep in mind that bitwise operation instructions cannot be used when optimization is not performed (`optimize=0`). Also, since the bitwise operation instruction has an access width of 8 bits, keep in mind that bitwise operation instructions cannot be used for variables declared as volatile whose access width is other than that of `signed` / `unsigned char` types.

Example 2-8

Source Program

```
typedef union {
    unsigned char BYTE;
        struct{
            unsigned char     B0:1;
            unsigned char     B1:1;
            unsigned char     B2:1;
            unsigned char     B3:1;
            unsigned short    W4:1;
            unsigned short    W5:1;
            unsigned long     L6:1;
            unsigned long     L7:1;
        }BIT;
}REG;
```

```
volatile REG Reg;
void func1(void){
        Reg.BIT.B2 = 1;
}
void func2(void){
        Reg.BIT.W4 = 1;
}
REG Reg2;
void func3(void){
        Reg2.BIT.W4 = 1;
}
```

SH-2

```
_func1:
  MOV.L        L13+4,R6      ;Reg
  MOV.B        @R6,R0
  OR           #32,R0
  RTS
  MOV.B        R0,@R6

_func2:
  MOV.L        L13+4,R6      ;Reg
  MOV.W        L13,R5        ; H'8000
  MOV.W        @(2,R6),R0
  OR           R5,R0
  RTS
  MOV.W        R0,@(2,R6)

_func3:
  MOV.L        L13+8,R6      ;Reg2
  MOV.B        @(2,R6),R0
  OR           #128,R0
  RTS
  MOV.B        R0,@(2,R6)

L13:
          .DATA.W      H'8000
          .RES.W       1
          .DATA.L      _Reg
          .DATA.L      _Reg2
```

SH-2A -abs20

```
_func1:
  MOVI20       #_Reg,R2
  BSET.B       #5,@(0,R2)
  RTS/N

_func2:
  MOVI20       #_Reg,R6
  MOV.W        @(2,R6),R0
  MOVI20       #32768,R5   ; H'00008000
  OR           R5,R0
  RTS
  MOV.W        R0,@(2,R6)

_func3:
  MOVI20       #_Reg2,R2
  BSET.B       #7,@(2,R2)
  RTS/N
```

Bitwise operation instruction

Due to the volatile declaration, the access width is guaranteed, and bitwise operation instructions cannot be used.

Because no volatile declaration exists, no access width guarantee is needed, and bitwise operation instructions are used.

2.9 Branching Instruction without Delay Slots

Format: `RTS/N`, `RTV/N Rm`

This instruction has no delay slots, and is returned from a subroutine procedure. Since SH-2 only has instructions returning from subroutine procedures with delay slots, when no processing exists to be placed in the delay slot, NOP is placed there instead. Since the NOP instruction can be deleted when RTS/N is used, the code size can be decreased. Note that with the SHC compiler, the function return value is returned to the `R0` register, but when the RTV/N instruction is used, since return instruction from the subroutine procedure can contain a transfer of data to the `R0` register, the code size can be decreased. The compiler uses this instruction automatically, and there are no particular items of concern about during coding.

Example 2-9

Source Program

```
int func(int a){

   return a;

}
```

SH-2

```
_func:
   RTS
   MOV.L        R4,R0
```

SH-2A -abs20

```
_func:
   RTV/N        R4
```

2.10 Barrel Shift Instruction

Format: `SHAD Rm,Rn`, `SHLD Rm,Rn`

This instruction shifts the contents of the `Rn` general register arithmetically (`SHAD`) or logically (`SHLD`). `Rm` indicates the shift direction and number of bits to be shifted. Since SH-2 does not have a barrel shift instruction, shift calculation is performed by a run-time routine if the shift count is undetermined, or by a combination of fixed shift instructions if the shift count is set. When a barrel shift instruction is used, both run-time routine calls and multiple calls for fixed shift instructions become unnecessary, to improve size, speed, and efficiency. The compiler uses this instruction automatically, and there are no particular items of concern about during coding.

Example 2-10 (1)

Source Program

```
int A;

int func(int s){

   A = A << s;

}
```

SH-2

```
_func:
   STS.L        PR,@-R15
   MOV.L        L24,R5      ; _A
   MOV.L        L24+4,R2    ; __sftl
   MOV.L        @R5,R0
   JSR          @R2
   MOV          R4,R1
   LDS.L        @R15+,PR
   RTS
   MOV.L        R0,@R5

L24
   .DATA.L      _A
   .DATA.L      __sftl
```

Run-time routine call

SH-2A -abs20

```
_func:
   MOVI20       #_A,R6
   MOV.L        @R6,R2
   SHAD         R4,R2
   RTS
   MOV.L        R2,@R6
```

Barrel shift instruction

Example 2-10(2)

Source Program	SH-2	SH-2A -abs20

Source Program

```
int A;

int func(int s){

   A = A << 14;

}
```

SH-2

```
_func:
   MOV.L        L25,R5    ; _A
   MOV.L        @R5,R6
   SHLL8        R6
   SHLL2        R6
   SHLL2        R6
   SHLL2        R6
   RTS
   MOV.L        R6,@R5

L25
   .DATA.L      _A
```

SH-2A -abs20

```
_func:
   MOVI20       #_A,R5
   MOV.L        @R5,R2
   MOV          #14,R6    ; H'0000000E
   SHAD         R6,R2
   RTS
   MOV.L        R2,@R5
```

3. Features and Usage of the New SH-2A Architecture

The following explains how to use the register bank and TBR of the new SH-2A architecture.

3.1 Register Bank

SH-2A comes with a register bank for speeding up save and restore processing for performing interrupt processing. Figure 3-1 shows the register bank configuration.

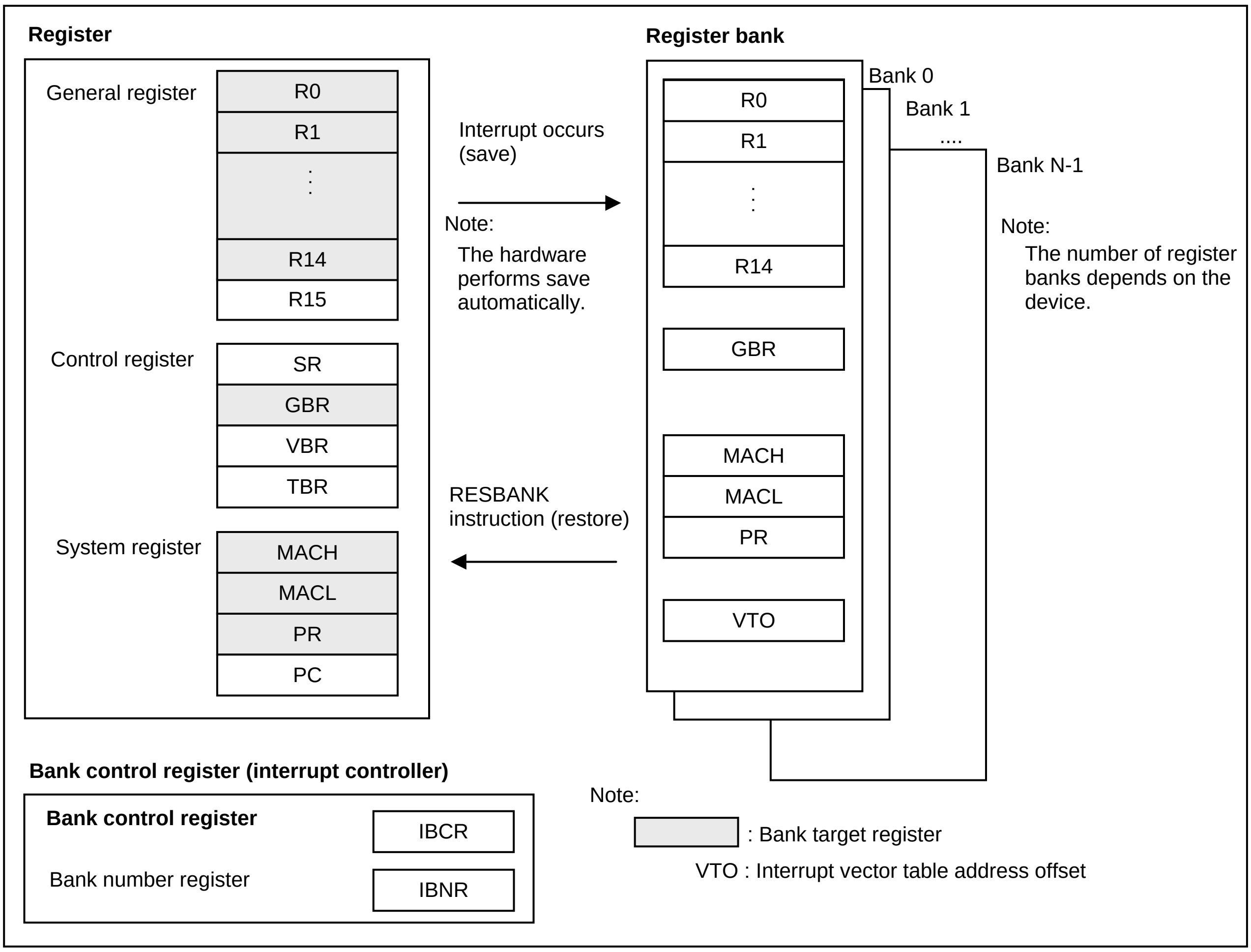


Figure 3-1 Overview and configuration of the register bank

When an interrupt occurs, the values in the register used by interrupt processing need to be saved and then restores. SH-2 used a stack to save and restore the necessary registers. This save and restore processing needs to be performed in software. When C is used, code to perform register save and restore is generated before and after the interrupt function (function specified by `#pragma interrupt`). SH-2A has a register bank system that uses hardware to speed up register save and restore. This register bank can be used to improve interrupt response speed significantly.

To use the register bank, both the bank control register settings and interrupt function specification need to correspond to the register bank.

For the bank control register settings, the bank number register (IBNR) and bank control register (IBCR) of the bank control register need to have the register bank to be enabled. For the actual values set, see *Register Bank* in the hardware documentation.

For the interrupt function specification, the register bank specification `resbank` needs to be performed for the interrupt function specification `#pragma interrupt`. For interrupt functions with `resbank` specified, instead of code that saves the register to the stack being generated at the start of the function, code that issues the `resbank` instruction is generated before the RTE instruction returns. The `resbank` specification is not performed for interrupt functions by the sample startup routine automatically generated by HEW for SH-2A. As such, an interrupt function that does not use a register bank (the same as SH-2) is generated instead. To use the register bank functionality for SH-2A, add the `resbank` specification, and set the bank control register. For detailed specifications for `#pragma interrupt`, see *10.3.1 #pragma extension* in the Compiler User Manual.

Example 3-1

SH-2A resbank not specified -abs20

```
C source code
#pragma interrupt (func)
void func(void);

unsigned int A;

void func(void){

        A = 0;

}
```

```
_func:
  MOV.L     R2,@-R15
  MOV.L     R3,@-R15
  MOV       #0,R2          ; H'00000000
  MOVI20    #_A,R3
  MOV.L     R2,@R3
  MOV.L     @R15+,R3
  MOV.L     @R15+,R2
  RTE
  NOP
```

Save/restore to/from stack

SH-2A resbank specified -abs20

```
C source code
#pragma interrupt (func2(resbank))
void func2(void);

unsigned int A;

void func(void){

        A = 0;

}
```

resbank specified

```
_func:
  Delete
  MOV       #0,R2          ; H'00000000
  MOVI20    #_A,R3
  MOV.L     R2,@R3
  Delete
  RESBANK
  RTE
  NOP
```

Deletion of save/restore to/from stack

3.2 Jump Table Base Register (TBR)

A jump table base register (TBR) has been added to the SH-2A control register. A TBR is a register that points to a function table placed in memory (TBR function table), and is used by the table reference subroutine call instruction `JSR/N @@(disp8,TBR)` (TBR register relative call). A TBR relative register call can be used to eliminate the need to read the function address register, to reduce code size and improve speed. Also, there are limitations on the function placement address range for function calls using 20-bit long immediate load (`MOVI20`) function, but not for subroutine calls using the TBR. Since the instruction length is 16 bits for table reference subroutine calls and 32 bits for 20-bit long immediate load, function calls using the TBR relative register call have a smaller code size.

To perform a function call using TBR, use the `#pragma tbr` (*function-name*) specification or TBR option. Function calls for functions for which `#pragma tbr` (*function-name*) is specified are TBR relative register calls, so that if the function is defined, the function address is output in the `$TBR` section. This `$TBR` section is the TBR function table. Up to 255 functions can be specified in a single TBR function table. If more than 255 functions need to be used, `#pragma tbr` (*function-name* (`sn=`*section-name*)) is used. Since the addresses of functions specified by `#pragma tbr` (*function-name* (`sn=`*section-name*)) are output to the `$TBR`*section-name* section, multiple TBR function tables can be used. When a function with a different TBR function table is called, the TBR function table is placed in ROM according to the linker section specification. When the TBR option is used, all functions in the file are TBR relative register calls. For detailed specifications for `#pragma tbr`, see *10.3.1 #pragma extension* in the Compiler User Manual. For details about the TBR option, see *2.3.4 C/C++ compiler operation method - Object option*.

Example 3-2

Source Program

```
(Function definition section)
#pragma tbr (test)
#pragma tbr (test2 (sn=_2))

unsigned int A;

void test(void);
void test2(void);

void test(void){

    A = 0;
}

void test2(void){

    A = 0;
}
```

As an example, test and test1 are placed in different TBR function tables

Always include this when using set_tbr.

```
(Function call section)
#include <machine.h>

#pragma tbr (test)
#pragma tbr (test2 (sn=_2))

void test(void);
void test2(void);

void call(void)
{
        set_tbr(__sectop("$TBR"));
        test();
        set_tbr(__sectop("$TBR_2"));
        test2();
```

The same #pragma tbr as that for the caller must be specified

Sets the start address of the TBR function table in the TBR before the TBR relative register call

Since a function with a different TBR function table is called, the TBR table needs to be re-set

SH-2A -abs20

```
(Function definition section)
          .EXPORT       _A
          .EXPORT       _test
          .EXPORT       _test2
          .EXPORT       $_test
          .EXPORT       $_test2
_test:
          MOV           #0,R2
          MOVI20        #_A,R3
          MOV.L         R2,@R3
          RTS/N

_test2:
          MOV           #0,R2
          MOVI20        #_A,R3
          MOV.L         R2,@R3
          RTS/N
_A:
          .RES.L        1

          .SECTION      $TBR,DATA,ALIGN=4
$_test:
          .DATA.L       _test

          .SECTION      $TBR_2,DATA,ALIGN=4
$_test2:
          .DATA.L       _test2
```

The address for test is output to the $TBR section

The address for test2 is output to the $TBR_2 section

```
(Function call section)
_call:
          STS.L         PR,@-R15

          MOV.L         L12+2,R2     ; STARTOF $TBR
          LDC           R2,TBR

          JSR/N         @@($_test-(STARTOF $TBR),TBR)

          MOV.L         L12+6,R2     ; STARTOF $TBR_2
          LDC           R2,TBR

          JSR/N         @@($_test2-(STARTOF $TBR_2),TBR)

          LDS.L         @R15+,PR
          RTS/N
L12:

          .RES.W        1
          .DATA.L       STARTOF $TBR
          .DATA.L       STARTOF $TBR_2
```

TBR setting

TBR relative register call

Website and Support <website and support,ws>

Renesas Technology Website

http://japan.renesas.com/

Inquiries

http://japan.renesas.com/inquiry

csc@renesas.com

Revision Record <revision history,rh>

Rev.	Date	Description	
		Page	Summary
1.00	Jun.01.07	—	First edition issued