Evaluation board/kit and Development tool important notice

1. This evaluation board/kit or development tool is designed for use for engineering evaluation, demonstration, or development purposes only. Do not use it for other purposes. It is not intended to meet the requirements of design for finished products.

2. This evaluation board/kit or development tool is intended for use by an electronics engineer and is not a consumer product. The user should use it properly and in a safe manner. Seiko Epson does not assume any responsibility or liability of any kind of damage and/or fire caused by the use of it. The user should cease to use it when any abnormal issue occurs even during proper and safe use.

3. The part used for this evaluation board/kit or development tool may be changed without any notice.

NOTICE

No part of this material may be reproduced or duplicated in any form or by any means without the written permission of Seiko Epson. Seiko Epson reserves the right to make changes to this material without notice. Seiko Epson does not assume any liability of any kind arising out of any inaccuracies contained in this material or due to its application or use in any product or circuit and, further, there is no representation that this material is applicable to products requiring high level reliability, such as, medical products. Moreover, no license to any intellectual property rights is granted by implication or otherwise, and there is no representation or warranty that anything made in accordance with this material will be free from any patent or copyright infringement of a third party. When exporting the products or technology described in this material, you should comply with the applicable export control laws and regulations and follow the procedures required by such laws and regulations. You are requested not to use, to resell, to export and/or to otherwise dispose of the products (and any technical information furnished, if any) for the development and/or manufacture of weapon of mass destruction or for other military purposes.

All brands or product names mentioned herein are trademarks and/or registered trademarks of their respective companies.

©SEIKO EPSON CORPORATION 2015, All rights reserved.
Table of Contents

1. Overview ........................................................................................................................................... 1
   1.1 Specification Outline .......................................................................................................................... 1
2. Architecture ............................................................................................................................................ 2
   2.1 Pipeline Stage .................................................................................................................................... 2
   2.2 Register Configuration ....................................................................................................................... 3
      2.2.1 Overview .................................................................................................................................... 3
      2.2.2 Address Registers (A0, A1, A2, ZR, XR, YR) .............................................................................. 3
      2.2.3 Register Shared for Address and Data (DR) .............................................................................. 3
      2.2.4 Data Registers (TZ, XDR, YDR, ZDR, ACR) ............................................................................. 4
      2.2.5 Working Address Register (WA) .............................................................................................. 4
      2.2.6 Special Address Registers (CBC0, CBC1, BRC0, BRC1, CBREF0, CBREF1) ......................... 4
      2.2.7 Operation Control Registers (ST, CD, SA, SS, RE, SC) ......................................................... 4
      2.2.8 Loop Control Registers (LPC0, LPC1) ...................................................................................... 4
3. Program Development Environment ...................................................................................................... 5
   3.1 Overview .......................................................................................................................................... 5
   3.2 How to Describe FSA Program .......................................................................................................... 8
      3.2.1 FSAREG Structure ....................................................................................................................... 8
      3.2.2 Checking FSA Status ................................................................................................................ 8
      3.2.3 Setting Program Counter (PC) .................................................................................................. 8
      3.2.4 Setting Forward-From and Forward-To Addresses and Number of Words to Be Transferred ... 9
      3.2.5 Kicking FSA off ......................................................................................................................... 9
      3.2.6 Describing FSA Program ......................................................................................................... 9
      3.2.7 Notifying of Completion of FSA Program .................................................................................. 9
   3.3 Low Level API Function .................................................................................................................... 10
   3.4 Handling Memory Accessed by FSA ................................................................................................. 11
      3.4.1 Definition of Constant Array .................................................................................................... 11
      3.4.2 Definition of Working Memory .............................................................................................. 11
      3.4.4 Referring to Constant Array and Working Memory ............................................................... 11
   3.5 Temporary Working Memory of FSA ............................................................................................... 12
   3.6 Compilation and Link ....................................................................................................................... 13
      3.6.1 Compiling FSA Program Part .................................................................................................... 13
      3.6.2 Compiling Other than FSA Program Part .................................................................................. 13
      3.6.3 Link ........................................................................................................................................... 14
   3.7 Constructing Emulation Environment .............................................................................................. 15
      3.7.1 Preparation .................................................................................................................................. 15
      3.7.2 Preprocess and Postprocess for Emulation .............................................................................. 15
      3.7.3 Profiling Report ....................................................................................................................... 19
      3.7.4 Log Information Output ......................................................................................................... 20
      3.7.5 Error and Warning Messages ............................................................................................... 21
   3.8 C17 + FSA Actual Machine Debugging Environment ................................................................. 23
      3.8.1 Overview .................................................................................................................................... 23
      3.8.2 Execution Procedures .............................................................................................................. 23
      3.8.3 How to Incorporate Application to Be Debugged .................................................................... 24
      3.8.4 Example of Using FSA-GDB ................................................................................................. 24
      3.8.5 Precautions and Restrictions ................................................................................................. 26
4. Instruction Set Specification .................................................................................................................. 27
5. Register Specification and Low Level API Function

5.1 Register Map

5.2 CTR Register

5.3 PC Register

5.4 A0, A1, A2, ZR, XR, YR Register

5.5 DR Register

5.6 WA Register

5.7 CBCR Register

5.8 BRCR Register

4.1 Instruction Set List

4.2 Address Operation Instruction

4.3 Data Operation Instruction

4.4 Right-Shift, Rounding, Saturation Process Control Instructions

4.5 Control Instruction for Conditional Execution

4.6 Program Loop Instruction

4.7 Operation Instruction Using Loop Control Register

4.8 Circular Buffer Addressing Control Instruction

4.9 Bit-Reverse Addressing Control Instruction

4.10 Special Instruction

4.11 Directive

4.12 Delay Slots Required for Avoiding Data Hazard

4. Operation Instruction Using Loop Control Register
5.9 LPCR Register ................................................................. 54
  5.9.1 Register Specification ............................................. 54
  5.9.2 API function ...................................................... 54

5.10 OPPR Register .......................................................... 55
  5.10.1 Register Specification ........................................ 55
  5.10.2 API function .................................................. 56

5.11 TZ Register ............................................................... 56
  5.11.1 Register Specification ..................................... 56
  5.11.2 API function .................................................. 56

5.12 VerInfo Register .......................................................... 57
  5.12.1 Register Specification ..................................... 57
  5.12.2 API function .................................................. 57

5.13 BAR Register .............................................................. 58
  5.13.1 Register Specification ..................................... 58
  5.13.2 API function .................................................. 58

5.14 IFR Register ............................................................... 59
  5.14.1 Register Specification ..................................... 59
  5.14.2 API function .................................................. 59

5.15 XOAR Register ............................................................ 60
  5.15.1 Register Specification ..................................... 60
  5.15.2 API function .................................................. 60

5.16 YOAR Register ............................................................ 61
  5.16.1 Register Specification ..................................... 61

5.17 ZOAR Register ............................................................ 62
  5.17.1 Register Specification ..................................... 62
  5.17.2 API function .................................................. 62

5.18 XDR Register .............................................................. 63
  5.18.1 Register Specification ..................................... 63
  5.18.2 API function .................................................. 63

5.19 YDR Register .............................................................. 64
  5.19.1 Register Specification ..................................... 64
  5.19.2 API function .................................................. 64

5.20 ZDR Register .............................................................. 65
  5.20.1 Register Specification ..................................... 65
  5.20.2 API function .................................................. 65

5.21 ACR Register .............................................................. 66
  5.21.1 Register Specification ..................................... 66
  5.21.2 API function .................................................. 66

Revision History ........................................................................ 67
1. Overview

FSA (Flexible Signal Processing Accelerator) is a new type of accelerator that enables you to flexibly modify process details by using micro-codes. FSA is applicable to entire product-sum operation based digital signal processing. FSA has the following features:

- **Light load on host CPU**
  Since FSA can accelerate fairly large amount of processes, the processing loads of host CPU can be remarkably reduced.

- **Reduced gates, memory saving, and high performance**
  The instruction set and architecture are adopted that have focused on the product-sum operation based digital signal processing.
  - 2-input and 1-output multi-data bus architecture
  - Parallel processing of data and address operations
  - Single cycle product-sum operation
  - Up to three modes of hardware saturation processing
  - Program/loop control without overhead
  - Execution of conditional instruction and conditional loop break
  - Automatic rounding according to arithmetic right-shift amount
  - Circular buffer and bit-reverse addressing

- **Excellent adaptability to target system**
  The data word length, accumulator bit length, multiplier configuration, and others are handled as parameters. They can be flexibly adapted to signal processing of target application from image processing to audio processing. Also, they can support AHB bus as the bus I/F.

1.1 Specification Outline

<table>
<thead>
<tr>
<th>Item</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction set</td>
<td>32bit fixed length, 22 instructions</td>
</tr>
<tr>
<td>Register set</td>
<td>Address register x 6, register shared for address and data x 1, data register x 4, and accumulator x 1</td>
</tr>
<tr>
<td>Address space</td>
<td>Up to 2MB, 21bit space</td>
</tr>
<tr>
<td>Bus</td>
<td>Program (32bit width), data (16 to 32bit width) x 3 (two for reading, one for writing)</td>
</tr>
<tr>
<td>Interrupt output</td>
<td>2 (IRQ1 for debugging &amp; monitoring)</td>
</tr>
</tbody>
</table>

Figure 1.1  FSA’s I/O
2. Architecture

2.1 Pipeline Stage

FSA has up to eight pipeline stages.

<table>
<thead>
<tr>
<th>IF</th>
<th>DE</th>
<th>MA</th>
<th>DR</th>
<th>E1</th>
<th>E2</th>
<th>E3</th>
<th>WB</th>
</tr>
</thead>
</table>

IF: Instruction fetch  
DE: Instruction decode, address calculation  
MA: Memory access to X bus and Y bus  
DR: Data read from X bus and Y bus  
E1: Multiplication  
E2: Addition & subtraction, accumulation  
E3: Shift, saturation processing  
WB: Write to Z bus

The operation pipeline has three stages: E1 to E3 stages.  
(see the figure at the right)

![Operation Pipeline Diagram](image-url)
2.2 Register Configuration

2.2.1 Overview

2.2.2 Address Registers (A0, A1, A2, ZR, XR, YR)

A0, A1, A2, ZR, XR, and YR are address registers. AT is 2bit field for holding address attribute (long, short, or uchar type). The address attribute controls the address register update and the size for reading/writing data. The bit width of address registers is configurable. (up to 21bit: 2MB space)

2.2.3 Register Shared for Address and Data (DR)

The DR register can store both address and data. The DR register also has the AT field. The bit width is the same as other data registers.
2. Architecture

2.2.4 Data Registers (TZ, XDR, YDR, ZDR, ACR)

ACR is an accumulator register. The bit width of each register is configurable. (The bit width of ACR is 72bit by default and the bit width of other data registers is 32bit.)

2.2.5 Working Address Register (WA)

WA register gets a working area for temporary use, similar to stack. The address indicated by the WA register indicates working area. Therefore, the address indicated by the WA register subtracted by the size to be used as the working area represents the beginning address of the area. The WA register is read only and its value is preset by the hardware.

2.2.6 Special Address Registers (CBC0, CBC1, BRC0, BRC1, CBREF0, CBREF1)

These registers control circular buffer and bit-reverse.

2.2.7 Operation Control Registers (ST, CD, SA, SS, RE, SC)

These registers control arithmetic right-shift, saturation processing, execution of conditional instruction and others.

2.2.8 Loop Control Registers (LPC0, LPC1)

These registers indirectly specifies the loop count.
3. Program Development Environment

3.1 Overview

The FSA program development is performed in the emulation environment on PC. The emulation environment has the following features:

- The FSA program and host program controlling the FSA program can be described in a single source file.
- The FSA program can be debugged by using existing favorite program development environment including Visual C++.
- The FSA program and host program can be seamlessly debugged in a single debugging environment.
- The operation of object codes was checked in the emulation environment so that the object codes for target system can be generated.
- The FSA object codes can be incorporated into the host's object file by using host CPU's linker.
- Similar to actual hardware, the wait cycle can be estimated that may occur because of access conflict to the same memory and bank.

The FSA emulation environment can be constructed by incorporating the FSA's behavior model library (fsabhv.lib) into existing program development environment including Visual C++. Describe FSA programs in inline format in the host program. FSA programs described in inline format make it possible to set breakpoints and run stepwise. Then, FSA's internal state can be shown on another window by using the debug monitoring application (fsadbm.exe). Also, the FSA programs described in inline format can be converted into object codes by dedicated tool (fsac.exe) and linked with object codes of the host program.

![Figure 3.1 FSA Program Development Environment](image)

- fsabhv.lib (fsabhv.dll)
  fsabhv.lib is a behavior model library that can provide the FSA emulation environment on PC. The behavior model can provide following functions for the emulation:
  - fsa function to describe FSA programs inline format in the C/C++ language source codes
  - Emulation function that has the same APIs as low level API functions to read and write FSA registers

The behavior model library can contain header files listed in the table below. These header files contain codes for target system. The program can be compiled for emulation by specifying _FSABHV during compilation. When emulating programs, make sure to specify _FSABHV during compilation. Note that only the fsa.h file should be included explicitly in all user programs.
3. Program Development Environment

Table 3.1  Header Files Included in Behavior Model

<table>
<thead>
<tr>
<th>Filename</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fsa.h</td>
<td>Defines low level API functions and constant parameters.</td>
</tr>
<tr>
<td>fsabhv.h</td>
<td>Declares library functions for emulation and defines constant parameters. They are included indirectly by fsa.h.</td>
</tr>
<tr>
<td>fsamac.h</td>
<td>Defines macro functions to describe instructions. They are included indirectly by fsa.h.</td>
</tr>
<tr>
<td>fsasys.h</td>
<td>Defines macro functions that are dependent on the host CPU. They are included indirectly by fsa.h.</td>
</tr>
</tbody>
</table>

- **fsac.exe**
  Puts FSA programs described in inline format into the C/C++ source files and converts them into standard GNU assembler files.

- **fsadbm.exe**
  This debug monitoring application shows the FSA's internal state on another window based on the interprocess communication with behavior model library.

![Debug Monitoring Application Overview](image-url)

In general, the number of wait cycles from previous breakpoint to current breakpoint is displayed. The error messages below are shown upon error occurrence.

- Source file name:Line No: reserved instruction error!
- Source file name:Line No: address error!
- Source file name:Line No: memory data hazard error!
- Source file name:Line No: DR data hazard error!
- Source file name:Line No: DR shift hazard error!
- Source file name:Line No: accumulator overflow!
- Source file name:Line No: loop nesting error!

Note: For more information, refer to 3.7.5 Error and Warning Messages
3. Program Development Environment

- fsamw.exe
  This memory window application shows the memory details at the specified address on another window based on the interprocess communication with behavior model library. The memory address can be displayed whose memory is mapped (described in 3.7.2 Preprocess and Postprocess for Emulation).

![Figure 3.3 Memory Window Application Overview](image-url)
3. Program Development Environment

3.2 How to Describe FSA Program

The codes below are a sample DMA program using FSA. The FsaDMA function is a simple DMA function and transfers the number of words specified by iSize from the address specified by pSrc to the address specified by pDst. This section describes how to describe the FSA program and build flow based on this sample program.

Sample (fsadma.c)

```c
#include "fsa.h" // Defines low level API functions and parameters.

void FsaDMA(FSAREG *pFsaReg, long *pIDst, long *pSrc, int iSize)
{
    extern unsigned long aulDMACode[]; // There is not any entity in the array and emulation of DMA programs.
    wait_fsa_finish(pFsaReg); // Macro function described in fsasys.h. Wait if FSA is busy.
    FSASetPC(pFsaReg, aulDMACode); // Set PC
    FSASetA0(pFsaReg, pSrc); // Set the source address
    FSASetA1(pFsaReg, pIDst); // Set the destination address
    FSASetLPCR(pFsaReg, iSize-1); // Set the number of words to be transferred
    FSASetCTR(pFsaReg, FSA_IRQ0_CLR | FSA_RUN); // Kick FSA off
    fsa(pFsaReg, _LOOPx (_LP0_)) {
        fsa(pFsaReg, _MOVA_ 1, _A1( 1), _A0( 1));
    }
    fsa(pFsaReg, _EXIT _ie0 _wait);
    wait_fsa_finish(pFsaReg); // Describe this in order to return from this function after DMA is finished.
}
```

3.2.1 FSAREG Structure

The functions controlling FSA must always have the pointer of the FSAREG structure as an argument. The pointer of the structure is FSA register address for the target system. For the emulation, the pointer is a handler structure for the behavior model. (The handler structure of behavior model can be obtained by using the FSABhvOpen function mentioned later.)

3.2.2 Checking FSA Status

The FSA status is checked by using the wait_fsa_finish function. When this function is called and the FSA status is busy, the host CPU stops. In principle, call the wait_fsa_finish function before calling the FSASetPC function to confirm that FSA is ready and then set the program counter. Also, when waiting for the completion of the FSA process, such as when the host CPU refers to the FSA's process result, use the wait_fsa_finish function.

3.2.3 Setting Program Counter (PC)

Set the address of the DMA program executed by FSA by using the low level API function, the FSASetPC function.

```c
extern unsigned long aulDMACode[]; // Array of DMA program
wait_fsa_finish(pFsaReg); // Check the FSA status
FSASetPC(pFsaReg, aulDMACode); // Set PC
```

As mentioned above, declare the program address as global array (aulDMACode). According to the build flow described later, the FSA program is assigned to aulDMACode.

In the emulation, the FSA program described in inline format is executed in the host program. Therefore, though this array does not have an entity, in order to consolidate the FSA program into codes for the target system, describe the array as mentioned above. A link error will not occur by defining _FSABHV even if aulDMACode does not have an entity.
### 3.2.4 Setting Forward-From and Forward-To Addresses and Number of Words to Be Transferred

Set address registers A0 and A1 each to the forward-from and forward-to addresses by using the FSASetA0 and FSASetA1 functions, low level API functions.

```c
FSASetA0(pFsaReg, plSrc); // Set the forward-from address
FSASetA1(pFsaReg, plDst); // Set the forward-to address
```

Specify the number of words to be transferred with the loop repeat count. In particular, use the FSASetLPCR function to set the LPC0 register to the number of words.

```c
FSASetLPCR(pFsaReg, iSize-1); // Set the number of words to be transferred
```

### 3.2.5 Kicking FSA off

When necessary registers are set up, kick FSA off by using the FSASetCTR function. In general, clear the IRQ0 status at the same time FSA is kicked off.

```c
FSASetCTR(pFsaReg, FSA_IRQ0_CLR | FSA_RUN); // Kick FSA off and clear the IRQ0 status
```

### 3.2.6 Describing FSA Program

Incorporate the FSA program in inline format into the host program on an instruction basis by using fsa functions with the specification shown below.

```c
fsa(pFsaReg, _LOOPx (_LP0_)) { // Execute the _MOVX instruction below as many times as specified by the LPC0 register
    fsa(pFsaReg, _MOVX (1, _A1( 1), _A0( 1)));
}
fsa(pFsaReg, _EXIT _ie0 _wait);
```

Note that, in the build flow mentioned later, the codes passed as the arguments of fsa functions are recognized as instructions and their object codes are generated.

### 3.2.7 Notifying of Completion of FSA Program

The _EXIT instruction notifies the host CPU of the completion of FSA program. In general, the host CPU is notified of through IRQ0 by specifying _ie0 option. The _wait option defers the execution of the _EXIT instruction until FSA finishes writing memory. If the host CPU may read the data immediately after FSA wrote the data, in principle, specify the _wait option.

```c
fsa(pFsaReg, _EXIT _ie0 _wait); // Set IRQ0. However, wait until the writing of memory is completed
```
3. Program Development Environment

3.3 Low Level API Function

Table 3.2 lists the low level API functions. These API functions are available for use in both the target system and emulation environment. For more information on the function specification, refer to 5. Register Specification and Low Level API Function.

<table>
<thead>
<tr>
<th>Low-level API functions</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>FSAGetCTR, FSASetCTR</td>
<td>Reads/writes the control register</td>
</tr>
<tr>
<td>FSASetMode</td>
<td>Sets the operation mode</td>
</tr>
<tr>
<td>FSAGetPC, FSASetPC, FSASetOnlyPC</td>
<td>Reads/writes the program counter</td>
</tr>
<tr>
<td>FSAGetA0, FSASetA0</td>
<td>Reads/writes the address register A0</td>
</tr>
<tr>
<td>FSAGetA1, FSASetA1</td>
<td>Reads/writes the address register A1</td>
</tr>
<tr>
<td>FSAGetA2, FSASetA2</td>
<td>Reads/writes the address register A2</td>
</tr>
<tr>
<td>FSAGetZR, FSASetZR</td>
<td>Reads/writes the address register ZR</td>
</tr>
<tr>
<td>FSAGetXR, FSASetXR</td>
<td>Reads/writes the address register XR</td>
</tr>
<tr>
<td>FSAGetYR, FSASetYR</td>
<td>Reads/writes the address register YR</td>
</tr>
<tr>
<td>FSAGetDR, FSASetDR, FSAGetDRDat, FSASetDRDat</td>
<td>Reads/writes the register DR shared for address and data</td>
</tr>
<tr>
<td>FSAGetWORK</td>
<td>Reads the working address</td>
</tr>
<tr>
<td>FSAGetCBCR, FSASetCBCR</td>
<td>Reads/writes the circular/buffer control register</td>
</tr>
<tr>
<td>FSAGetBRCR, FSASetBRCR</td>
<td>Reads/writes the bit-reverse control register</td>
</tr>
<tr>
<td>FSAGetLPCR, FSASetLPCR</td>
<td>Reads/writes the register indirect loop control register</td>
</tr>
<tr>
<td>FSAGetOPPR, FSASetOPPR</td>
<td>Reads/writes the operation control parameter/register</td>
</tr>
<tr>
<td>FSAGetTZ, FSASetTZ</td>
<td>Reads/writes the data register TZ</td>
</tr>
<tr>
<td>FSAGetVerInfo</td>
<td>Reads the version information</td>
</tr>
<tr>
<td>FSAGetBAR, FSASetBAR</td>
<td>Reads/writes the break address register</td>
</tr>
<tr>
<td>FSAGetIFR</td>
<td>Reads the instruction fetch register</td>
</tr>
<tr>
<td>FSAGetXOAR</td>
<td>Reads the X bus address register</td>
</tr>
<tr>
<td>FSAGetYOAR</td>
<td>Reads the Y bus address register</td>
</tr>
<tr>
<td>FSAGetZOAR</td>
<td>Reads the Z bus address register</td>
</tr>
<tr>
<td>FSAGetXDR</td>
<td>Reads the X bus data register</td>
</tr>
<tr>
<td>FSAGetYDR</td>
<td>Reads the Y bus data register</td>
</tr>
<tr>
<td>FSAGetZDR</td>
<td>Reads the Z bus data register</td>
</tr>
<tr>
<td>FSAGetACR0, FSAGetACR1, FSAGetACR2</td>
<td>Reads the accumulator register</td>
</tr>
</tbody>
</table>
3. Program Development Environment

3.4 Handling Memory Accessed by FSA

3.4.1 Definition of Constant Array

Define a constant array accessed directly by FSA as global array in the host program. (Do not specify the static storage class). As shown in the following example, append the keyword __FSA_CONSTANTS__. This keyword sets the section name of the array defined to .fsacons. Use the linker script to map constant array to desired addresses by using this section name.

```
const long alFsaConst[] __FSA_CONSTANTS__ = {
    0x40000000, 0x00000000, 0x3B20D79E, 0x187DE2A6,
    0x2D413CCC, 0x2D413CCC, 0x187DE2A6, 0x3B20D79E,
    0x00000000, 0x40000000, 0xE7821D5A, 0x3B20D79E,
    0xD2BEC334, 0x2D413CCC, 0xC4DF2862, 0x187DE2A6
};
```

3.4.2 Definition of Working Memory

Define a working memory accessed directly by FSA as global variable in the host program. (Do not specify the static storage class). As shown in the following example, append the keyword __FSA_WORK__. This keyword sets the section name of the variable defined to .fsawork. Use the linker script to map the variable to desired address by using this section name.

```
long aulFsaCircularBuf[16] __FSA_WORK__ __FSA_ALIGNED__(64);
```

3.4.4 Referring to Constant Array and Working Memory

To set address registers to immediates in the FSA program, use the _SETAD instruction. However, when the address specified is global symbol, pass the address to the _SETAD instruction indirectly through the __FSA_GLOBAL__ macro function.

```
extern const long alFsaConst[];
...
fsa(pFsaReg, _SETAD (_A0_, __FSA_GLOBAL__(alFsaConst), _long));
```

```
extern unsigned char aucFsaWork[];
...
fsa(pFsaReg, _SETAD (_A1_, __FSA_GLOBAL__(aucFsaWork), _uchar));
```
3. Program Development Environment

3.5 Temporary Working Memory of FSA

When performing some process, the memory may be required as temporary working space. FSA can get this working memory directly without the intervention of host CPU. Assign the area just below the address indicated by the WA register as the working memory to the target system. (FSA system design rules) Therefore, for example, to set the A0 register to the address of 16word working memory, describe as follows by using the _MOVAD instruction.

Example to set working memory address

```c
fsa(pFsaReg, _MOVAD(_A0_, _WA( -16), _int));
```
3.6 Compilation and Link

3.6.1 Compiling FSA Program Part

Compile the FSA programs described in inline format by using the fsa function in the host program in the flow shown below. First, resolve #define macro with the preprocessor in the host CPU. Here, define _FSA_COMPILE. Next, extract and convert the FSA program part using the dedicated tool fsac into standard GNU assembler file. Finally, convert the file using “int” in the host CPU into an object file. The information for resolving undefined symbols in the FSA program is appended to the fsa_symbols.def file each time fsac is executed. This file is used at the time of link.

The data in the object file obtained is allocatable array data with the section name .fsacode and can be linked with other object files in the host CPU.

Figure 3.4 Compilation Flow of FSA Program

The pipe connection as shown below is available for compiling the FSA program part.

```
host-gcc -E fsadma.c -D_FSA_COMPILE -I$(FSA_DIR)/include | fsac [-g] [-32bit] | host-as -o fsadma_fsa.o
```

Note: FSA_DIR is an environment variable to be set by installing FSA 2.0 development environment.
In the line above, the path of fsa.h, fsamac.h, fsabhv.h, and fsasys.h is specified.
Note: Specify -32bit option when the host CPU is 32bit
Note: To generate the debugging information, add -g option to fsac

3.6.2 Compiling Other than FSA Program Part

Compile other than FSA program part with the compiler in the host CPU without defining _FSABHV and _FSA_COMPILE. Since the FSA program part described in inline format becomes blank, its codes are not generated. Also, compile the constant arrays referred to directly by FSA as usual constant arrays in this compilation path.
3. Program Development Environment

3.6.3 Link

As mentioned above, two object files of the FSA program and host program are generated from the source program including the FSA program through two compilation paths. The object files obtained in this manner can be linked by the linker in the host CPU. Then, specify fsa_symbols.def generated by fsac with -R option on the linker's command line.

Since the section name of the FSA program array is .fsacode and that of constant array addressed directly by FSA is .fsacons, the object codes can be mapped to the address specified by the linker script by using these section names.

```
OBJJS=    target.o fsadma.o ...
FSAOBJS= fsadma_fsa.o ...

host-ld -T target.lds -R fsa_symbols.def -o $(OBJJS) $(FSAOBJS)
```

Figure 3.5 shows the whole build flow.
3. Program Development Environment

3.7 Constructing Emulation Environment

3.7.1 Preparation

To construct the emulation environment, perform the preparation below.

- Link the behavior model library (fsabhv.lib).
- Define _FSABHV in the compile option.
- Include fsa.h in source file where the behavior model functions are called.

3.7.2 Preprocess and Postprocess for Emulation

Furthermore, to run the FSA program in the emulation environment, the following processes specific to emulation environment are required.

![Emulation Process Flow Diagram]

1. Reserving area for FSAREG structure

Call the FSABhvOpen function as shown below to get the pointer to the FSAREG structure. In the target system, the address of the FSAREG structure is put in the FSA register, so use the data in the FSA register to reserve the area of the structure. In the emulation, the FSABhvOpen function reserves the area of the FSAREG structure. The FSABhvOpen function returns zero if the area was not reserved.

```c
FSAREG *pFsaReg;

if (0 == (pFsaReg = FSABhvOpen())) {
    // Failed to reserve area
}
```
3. Program Development Environment

2. Initialization

Then, call the FSABhvInit function to initialize the area. Specify the value put in WA register as second argument. The WA register has been set by hardware in advance in the target system. But in the emulation, set the WA register here. It is required to map necessary size of memory to the address indicated by the WA register.

```c
#define FSA_WORK_ADDR   0x10000
...
FSABhvInit(pFsaReg, FSA_WORK_ADDR);  // Set WA register to 0x10000
```

3. Memory mapping

As the memory space handled by the FSA hardware, the address range to be processed by 12 to 21 bits or so is assumed. The address generation on the inside of FSA behavior model copies after real hardware and the memory space handled by the FSA program is the same as that of actual FSA hardware. Since the world outside of the behavior model library is managed with 32-bit memory address, when exchanging addresses between inside and outside of the behavior model, it is necessary to convert the addresses. (Now, the address on the inside of behavior model is called a virtual address.)

In order to take measures for this problem, the mapping information for address conversion is generated at initialization, and the addresses are automatically converted in exchanges through low level API functions based on the mapping information.

In particular, call the FSABhvMemMap or FSABhvConstMap function to specify how to map the memory reserved on the outside of behavior model to the memory space in the behavior model.

![Figure 3.7 Schematic Diagram for Address Conversion](image)

As mentioned above, the address is automatically converted when referring to and setting the address through low level API functions. Avoid directly referring to or changing the member variables of FSAREG structure.
3. Program Development Environment

Use the FSABhvMemMap function when mapping the RAM area and the FSABhvComstMap function when mapping the ROM area. Note that the range between lower and upper limit addresses is up to 2MB. (The memory space accessed directly by FSA is up to 2MB.)

**FSABhvMemMap**

| Format | unsigned long FSABhvMemMap( void *p, unsigned long ulAddr, unsigned long ulSize, unsigned long ulConnect ) |

**Argument**

- **p**: Pointer to the memory area reserved. The area is reserved internally by passing a NULL pointer.
- **ulAddr**: Virtual address to be mapped
- **ulSize**: Size (byte) of memory to be mapped
- **ulConnect**: Specify access permission to the memory to be mapped. Specify the combination (logical OR) of following values.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FSA_XBUS</td>
<td>Allows the access from X bus.</td>
</tr>
<tr>
<td>FSA_YBUS</td>
<td>Allows the access from Y bus.</td>
</tr>
<tr>
<td>FSA_ZBUS</td>
<td>Allows the access from Z bus.</td>
</tr>
</tbody>
</table>

**Return Value**

- If successful, a 4-byte aligned value ulAddr+ulSize is returned. If failed, zero is returned.

**Description**

To map the already reserved memory, specify its pointer *p and set ulAddr to the FSA's virtual address to be mapped. The area is reserved internally based on ulSize by passing a NULL pointer to *p. (The latter is recommended) The pointer to memory internally reserved can be obtained by the FSABhvGetMemPtr function. The memory mapped by the FSABhvMemMap function is handled as a memory bank. In actual hardware, if multiple access requests to a memory bank occur at the same time, the instruction execution is deferred. In behavior model, the wait cycle because of access conflict can be estimated.

Also, set the ulConnect to the path to which the access is allowed. For example, when FSA_YBUS is specified, the access from Y bus is allowed. But, the access from other buses causes an address error.

The following code example describes the mapping of the memory bank configuration shown in Figure 3.8. It is not necessary to perform the mapping in ascending order of addresses. However, get the pointer to the memory reserved by the FSABhvGetMemPtr function after the mapping of all banks was finished.

```c
#define BASE_ADDR   0x00100000

unsigned long ulAddr = BASE_ADDR;
unsigned char *pucAddr;

ulAddr = FSABhvMemMap(0, ulAddr, 65536, FSA_XBUS | FSA_YBUS | FSA_ZBUS); // Bank0
ulAddr = FSABhvMemMap(0, ulAddr, 65536, FSA_XBUS | FSA_YBUS | FSA_ZBUS); // Bank1
ulAddr = FSABhvMemMap(0, ulAddr, 131072, FSA_XBUS | FSA_YBUS | FSA_ZBUS); // Bank2

pucAddr = (unsigned char*)FSABhvGetMemPtr(BASE_ADDR); // Extracting the memory pointer
```
3. Program Development Environment

![Memory and Bank Configuration Example](image)

**FSABhvConstMap**

Format: `unsigned long FSABhvConstMap( const void *p, unsigned long ulAddr, unsigned long ulSize )`

Argument:
- `p`: Pointer to constant array
- `ulAddr`: Virtual address to be mapped
- `ulSize`: Size (byte) of memory to be mapped

Return Value:
- If successful, a 4byte aligned value `ulAddr+ulSize` is returned. If failed, zero is returned.

Description:
- Map the read only memory for constant array by using the `FSABhvConstMap` function. The memory mapped with this function can be read only by the X bus and Y bus.

**FSABhvGetMemPtr**

Format: `void* FSABhvGetMemPtr(const unsigned long ulAddr)`

Argument:
- `ulAddr`: Virtual address

Return Value:
- If successful, the pointer corresponding to virtual address specified by `ulAddr` is returned. If failed, zero is returned.

Description:
- To get the pointer to the memory reserved by the `FSABhvMemMap` function, use this function. It is necessary to get the pointer after the mapping of all memories was finished.
4. Releasing Area for Handler Structure in Behavior Model

As a postprocess, call the FSABhvClose function to release the area for the FSAREG structure.

FSABhvClose(pFsaReg);

3.7.3 Profiling Report

The FSABhvClose function outputs the profiling report as shown below to the standard output. INST.CYCLE represents the number of instruction cycles, and REQ.CYCLE represents the number of request cycles including wait cycle because of the access conflict among the FSA's X-bus, Y-bus, and Z-bus. Therefore, the impact from FSA program load, access conflict with other module including host CPU, and others are not reflected. The function name indicated by FUNCTION represents the array name for FSA program set by the FSASetPC function.

<table>
<thead>
<tr>
<th>CYCLE PER FUNCTION</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>REQ.CYCLE (RE/IN)</td>
<td>INST.CYCLE</td>
<td>FUNCTION</td>
<td></td>
</tr>
<tr>
<td>510 (1.969)</td>
<td>259 =&gt; function0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111 (1.057)</td>
<td>105 =&gt; function1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>515 (1.000)</td>
<td>515 =&gt; function2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>56 (1.120)</td>
<td>50 =&gt; function3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7 (1.000)</td>
<td>7 =&gt; function4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5213 (1.000)</td>
<td>5211 =&gt; function5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3274 (1.228)</td>
<td>2666 =&gt; function6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13530 (1.488)</td>
<td>9093 =&gt; function7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3274 (1.228)</td>
<td>2666 =&gt; function8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1546 (1.404)</td>
<td>1101 =&gt; function9</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

TOTAL CYCLE PER FUNCTION

<table>
<thead>
<tr>
<th></th>
<th>CALL</th>
<th>FUNCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>357187</td>
<td>699</td>
<td>function0</td>
</tr>
<tr>
<td>1230939</td>
<td>12582</td>
<td>function1</td>
</tr>
<tr>
<td>359985</td>
<td>699</td>
<td>function2</td>
</tr>
<tr>
<td>9583980</td>
<td>356208</td>
<td>function3</td>
</tr>
<tr>
<td>4893</td>
<td>699</td>
<td>function4</td>
</tr>
<tr>
<td>290777079</td>
<td>365335</td>
<td>function5</td>
</tr>
<tr>
<td>2288526</td>
<td>699</td>
<td>function6</td>
</tr>
<tr>
<td>9457470</td>
<td>699</td>
<td>function7</td>
</tr>
<tr>
<td>2288526</td>
<td>699</td>
<td>function8</td>
</tr>
<tr>
<td>1079108</td>
<td>698</td>
<td>function9</td>
</tr>
</tbody>
</table>

TOTAL REQUIRED CYCLE: 317427693
3. Program Development Environment

3.7.4 Log Information Output

FSABhvSetMode(pFsaReg, FSA_LOGOUT);

Call the FSABhvSetMode function as shown above to output the log information from the call time as shown below to standard output.

Sample to output log information

```
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< FSA START >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
00087:12E04000  _SETAD  (_YR_, 0x004000, _long)
   YR = 0x004000 (long)
00092:812524C1  _MP     (1, _AC_, _DR_, _i1_)
   XDR=0x0007FFFF(32767)
   YDR=0x00000001(1)
   ACR=0x000000000000000000007FFF(32767)
00093:4125304F  _ADD    (1, _DR_, _i1_, _iM1_)
   XDR=0x00000001(1)
   YDR=0xFFFFFFFF(-1)
   DR=0x00000000(0)
00101:18FC0001  _SETCD  (_gez)
   ST=0
00102:01000001E  _LOOP   (1, 31)
   ------< LOOP NESTING: 0, REMAINING COUNT: 31 >-------------
00103:E13D6501  _MMAC   (1, _ST_, _TZ_, _YR(  1))
   XDR=0x000000000000000000007FFF(32767)
   YOAR=0x00040000, YDR=0x00000000(1073741824)
   ST=1
00104:E1252511  _MMAC   (1, _AC_, _TZ_, _YD_)
   XDR=0x000000000000000000007FFF(32767)
   YDR=0x00000000(0)
00105:412534D1  _ADD    (1, _DR_, _DR_, _YD_)
   XDR=0x00000000(0)
   YDR=0x00000000(0)
   DR=0x00000000(0)
   ------< LOOP NESTING: 0, REMAINING COUNT: 30 >-------------
00103:E13D6501  _MMAC   (1, _ST_, _TZ_, _YR(  1))
   XDR=0x000000000000000000007FFF(32767)
   YOAR=0x00040000, YDR=0x00000000(1073741824)
   ST=1
...

To stop the log information output, call the FSABhvSetMode function as shown below.

FSABhvSetMode(pFsaReg, FSA_NORMAL);
3. Program Development Environment

3.7.5 Error and Warning Messages

In the emulation environment, the behavior model outputs the error messages and warning messages below to the standard error output.

**Error message**

This section describes error messages. When one of these errors occurs, the behavior model cancels all subsequent emulation.

- **[File name]:[Line number]: reserved instruction error has occurred!**  
  The instruction indicated by the file name and line number caused a reserved instruction error.

- **[File name]:[Line number]: address error has occurred!**  
  The instruction indicated by the file name and line number accessed to the address not allocated.

- **[File name]:[Line number]: memory data hazard error has occurred!**  
  The instruction indicated by the file name and line number read the result of preceding operation instruction before the result was written into the memory.  
  Supplement: When reading the data written into memory, four instructions of delay is required. For more information, refer to 4.12. Delay Slots Required for Avoiding Data Hazard.

- **[File name]:[Line number]: the change of DR effects DR that is read by two previous instructions!**  
  The change of DR register caused by the instruction indicated by the file name and line number gives an impact on the reading of DR register caused by two previous instructions.  
  Supplement: When a data operation instruction takes DR register as a source operand, do not change the DR register with the _SETAD or _MOVAD instruction in two subsequent instructions. For more information, refer to 4.12 Delay Slots Required for Avoiding Data Hazard.

- **[File name]:[Line number]: the change of DR effects shift by DR in the previous four instructions!**  
  The change of DR register caused by the instruction indicated by the file name and line number gives an impact on the right-shift with DR register caused by four previous instructions.  
  Supplement: When performing the right-shift, a data operation instruction, with the DR register, do not change the DR register with the _SETAD or _MOVAD instruction in four subsequent instructions. For more information, refer to 4.12. Delay Slots Required for Avoiding Data Hazard.

- **[File name]:[Line number]: accumulator overflow has occurred!**  
  The ACR register overflowed with the data operation instruction indicated by the file name and line number.

- **[File name]:[Line number]: loop nesting error!**  
  The nesting exceeded the specified value with the _LOOP (or _LOOPx) instruction indicated by the file name and line number.  
  Supplement: When innermost program loop is two or less instructions, the nesting can be up to four levels. Otherwise, the nesting can be up to three levels.
3. Program Development Environment

Warning message
This section describes warning messages. Upon warning, the behavior model continues the emulation.

- [File name]:[Line number]: FSA has been kicked without checking the status!
The FSASetCTR function indicated by the file name and line number kicked FSA off without confirming ready state.
Supplement: When FSA is further kicked off without confirming the end of the program kicked off after kicking FSA off, this message is shown. In general, make sure to confirm the end of program with the wait_fsa_finish function.

- [File name]:[Line number]: _EXIT instruction wasn't executed in the last FSA program!
There is not the _EXIT instruction in last executed FSA program.
Supplement: This warning is not critical to the execution of emulation. However, it is critical for existing equipment system. Take measures quickly.

- [File name]:[Line number]: the instruction can't read the latest DR!
The data operation instruction indicated by the file name and line number cannot read the latest DR register.
Supplement: The previous instruction is updating the DR register. When reading this data, an instruction of delay is required. For more information, refer to 4.12. Delay Slots Required for Avoiding Data Hazard. However, when reading the DR register intentionally before the update, there is no problem.

- [File name]:[Line number]: the instruction can't read the latest TZ!
The data operation instruction indicated by the file name and line number cannot read the latest TZ register.
Supplement: Two previous instructions updated the TZ register. When reading this data, two instructions of delay are required. For more information, refer to 4.12. Delay Slots Required for Avoiding Data Hazard. However, when reading the TZ register intentionally before the update, there is no problem.

- [File name]:[Line number]: the instruction can't read the latest XD!
The data operation instruction indicated by the file name and line number cannot read the latest XD register.
Supplement: Two previous instructions updated the XD register. When reading this data, two instructions of delay are required. For more information, refer to 4.12. Delay Slots Required for Avoiding Data Hazard. However, when reading the XD register intentionally before the update, there is no problem.

- [File name]:[Line number]: the change of DR is in conflict with the proceeding instructions!
The update of DR register indicated by the file name and line number conflicts with the update of preceding data operation instruction.
Supplement: There may be an error in the program. Check the program.

- [File name]:[Line number]: loop nesting has reached four level!
The nesting reached to four levels with the _LOOP (or _LOOPx) instruction indicated by the file name and line number.
Supplement: There is no problem in normal operation, but the debugging will be disabled in actual machine system.
3. Program Development Environment

3.8 C17 + FSA Actual Machine Debugging Environment

3.8.1 Overview

The actual machine debugging is possible by using C17 + FSA implemented in FPGA, etc. The figure below shows the software related to physical connection diagram.

The part in blue represents the software.
- C17-GDB
  Debugger (gdb.exe) coming with GNU17.
- FSA-GDB
  Debugger (fsa-epson-elf-gdb.exe) for FSA.
- stub
  Library for controlling the reading/writing of memories and registers and the execution according to the direction from FSA-GDB.
- application
  User application including FSA programs for debugging. The way to incorporate the application will be described later.

3.8.2 Execution Procedures

Follow the procedures below to debug an FSA program.
1. Launch C17-GDB and perform the following initialization. Then, perform to continue the polling in step 5.
   file xxxx.elf
   target icd usb
   set $ttbr=0x000000
   load
c17 rst
   (refer to gdb-stub.cmd in the sample directory described later)

2. Launch FSA-GDB and perform the following initialization.
   set remote baud 115200
   set endian little
   set remote hardware-breakpoint-limit 1
   set can-use-hw-watchpoints 0
   (refer to .gdbinit in the sample directory described later)

3. FSA-GDB connects to the target.
   target remote COMx  (x may vary according to the environment.)
   file sample.elf  (Only when debugging symbols.)

4. Set breakpoints and others, if necessary.

5. Execute stepi or continue from FSA-GDB.
3. Program Development Environment

3.8.3 How to Incorporate Application to Be Debugged

This section describes how to incorporate an application to be debugged. When FSA 2.0 installer is executed, the sample directory for debugging actual machine are installed into the directory below by default.

C:/EPSON/FSA2.0/sample/c17debug/

For more information, refer to files in the sample directory.

Use the following files:
- boot.c : Defines the vector area, etc.
  The interrupts of UART Ch0 (16) and FSA IE1 (23) are used.
- main.c : Main function
- sample.c : Application to be debugged
  As a sample, the FSA program for division is used.
- c17stub.a : Archive for stub
- Makefile : Makefile for creating ELF file
- gdb-stub.lds : Linker script for creating ELF file

Call the followings at the beginning and end of main function.
  init_stub
  close_stub

3.8.4 Example of Using FSA-GDB

This section lists examples of using main GDB commands. For more information, refer to general GDB manual.

- Break point
  ex1: Sets a software breakpoint.
  break *0x105008
  ex2: Sets a hardware breakpoint.
  hbreak *0x105008
  ex3: Deletes the breakpoint 1.
  delete breakpoints 1
  ex4: Sets a conditional breakpoint.
  break *0x105008 if $DR == 3 (only when DR register is three)
  ex5: Shows breakpoints established.
  info breakpoints
  ex6: Sets a breakpoint with file name (file) and line number (num). (However, only after executing the file command.)
  break file:num
  ex7: Sets a breakpoint with the array name (aulFsaArray) for FSA program. (However, only after executing the file command.)
  break aulFsaArray

- Run
  ex1: Executes the program until next breakpoint.
  continue
  ex2: Executes the program by ignoring the breakpoint at a current position as many as N times.
  continue N
  ex3: Execute as many as an instruction.
  stepi

- Read/write register
  ex1: Displays all registers.
  info all-registers
  ex2: Displays CTR and PC registers.
  info registers CTR PC
  ex3: Sets PC to 0x1234.
  set $PC=0x1234
3. Program Development Environment

- Read/write memory
  ex1: Writes four bytes of data 0x12345678 into the address 0x10c000.
      set *(unsigned long*)0x10c000=0x12345678
  ex2: Writes a file into the address 0x10c000.
      restore file binary 0x10c000
  ex3: Displays 32 x 4 bytes from the address 0x10c000.
      x/32 0x10c000
  ex4: Dumps the data from the address 0x10c000 to 0x10c100 into a file.
      dump binary memory file 0x10c000 0x10c100

- Reverse assemble
  ex1: Reverse assembles the codes from 0x105000 to 0x105008.
      disassemble 0x105000 0x105008

- List
  ex1: Displays source codes using the array name (aulFsaArray) for FSA program. (However, only after
      executing the file command.)
      list aulFsaArray

- Watch point
  ex1: Breaks if the value at the address 0x10c000 is changed.
      watch *0x10c000

An abbreviation for each GDB command above is available. The list below shows examples.

continue  c
stepi    si
break    b
hbreak   hb
delete   d
info     inf
all-registers  al
dump     du
restore  res
disassemble  disas
list     l
watch    wa
3. Program Development Environment

3.8.5 Precautions and Restrictions

This section describes precautions and restrictions. Be aware of them.

- Since there is not the concept of function call, it's impossible to trace stacks, for example.
- The command step has the same operation as stepi.
- The available number of software breakpoints is up to 32.
- When stepi is executed at the end of loop, the loop seems to exit and next instruction seems to be carried out without returning back to the beginning of the loop. However, in fact, the loop is carried out normally.
- According to the FSA specification, when the deepest loop is an on-fetch loop in the multiple loop, up to four loops can be nested. But, it is impossible to debug the FSA program with such a loop implementation.
- The first stepi after C17-GDB startup or after reset behaves in the same manner as continue.
- When stepi is executed from just one before the FSA's _EXIT instruction, since the FSA instruction to be executed next is unknown, stepi behaves in the same manner as continue, as described above. Therefore, set a breakpoint at the address to be executed next, if needed.
- When stepi is executed from just one before the FSA's _EXIT instruction, PC seems to break twice at the same address.
- The breakpoint cannot be set at other than .fsacode section.
- Though the reserved instruction exception is detected, the debugging becomes impossible after that.
- The instruction continue or stepi must be executed from FSA-GDB at the first debugging step at least. If you want to run all processes without using FSA-GDB, delete init_stub and close_stub.
- Since the CTR register is put into debug mode in init_stub, the execution speed gets slow unexpectedly.
- FSA's IE0 interrupt is not used. Define an interrupt handler and then set the argument of init_stub to other than zero, if necessary.
- When the FSA's I/O area (from 0x5400) is shown in 32bit using the print instruction, upper and lower 16bits are shown with the same value on a 8bit basis. (hardware specification)
- The exceptions below cannot be detected that can be detected in the behavior model. (hardware specification)
- Address error exception, data hazard, overflow, depth of loop nest
- rwatch, awatch, and catchpoint are not supported.
- When stepi is executed after the close_stub call, the control may not return. This is because the FSA's PC register is just pointing invalid address. Set PC, address register, and others in FSA-GDB, if necessary, to continue the debugging.
- cygwin whose version is different from GNU17 cannot be executed at the same time.
### 4. Instruction Set Specification

#### 4.1 Instruction Set List

<table>
<thead>
<tr>
<th>Address operations</th>
<th>Opecode</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>_SETAD</td>
<td>Sets address register</td>
<td></td>
</tr>
<tr>
<td>_MOVAD</td>
<td>Copies address register</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Arithmetic operations</th>
<th>Opecode</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>_ADD</td>
<td>Addition</td>
<td></td>
</tr>
<tr>
<td>_SUB</td>
<td>Subtraction</td>
<td></td>
</tr>
<tr>
<td>_MP</td>
<td>Multiplication</td>
<td></td>
</tr>
<tr>
<td>_MAC</td>
<td>Multiplication and accumulation</td>
<td></td>
</tr>
<tr>
<td>_MMAC</td>
<td>Multiplication and minus-accumulation</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Logical operations</th>
<th>Opecode</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>_AND</td>
<td>AND</td>
<td></td>
</tr>
<tr>
<td>_OR</td>
<td>OR</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operation controls</th>
<th>Opecode</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>_SETSC</td>
<td>Sets shift amount and saturation controls</td>
<td></td>
</tr>
<tr>
<td>_MOVSC</td>
<td>Operates shift amount</td>
<td></td>
</tr>
<tr>
<td>_SETCD</td>
<td>Sets conditional controls</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Program loop controls</th>
<th>Opecode</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>_LOOP</td>
<td>Immediate loop</td>
<td></td>
</tr>
<tr>
<td>_LOOPx</td>
<td>Register indirect loop</td>
<td></td>
</tr>
<tr>
<td>_SETLP</td>
<td>Sets loop control register</td>
<td></td>
</tr>
<tr>
<td>_MOVLP</td>
<td>Operates loop control register</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Special instructions</th>
<th>Opecode</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>_NOP</td>
<td>NOP</td>
<td></td>
</tr>
<tr>
<td>_EXIT</td>
<td>Program termination, interrupt control</td>
<td></td>
</tr>
<tr>
<td>_BREAK</td>
<td>Break instruction for debug</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Circular buffer and bit-reverse controls</th>
<th>Opecode</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>_SETCB</td>
<td>Sets circular buffer control</td>
<td></td>
</tr>
<tr>
<td>_MOVCB</td>
<td>Operates circular buffer control</td>
<td></td>
</tr>
<tr>
<td>_SETBR</td>
<td>Sets bit-reverse control</td>
<td></td>
</tr>
<tr>
<td>_MOVBR</td>
<td>Operates bit-reverse control</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Pseudo-instruction</th>
<th>Opecode</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>_MOVX</td>
<td>Copies X-bus input data</td>
<td></td>
</tr>
<tr>
<td>_MOYY</td>
<td>Copies Y-bus input data</td>
<td></td>
</tr>
<tr>
<td>_MOVAC</td>
<td>Copies accumulator</td>
<td></td>
</tr>
<tr>
<td>_SETZR</td>
<td>Zero clear</td>
<td></td>
</tr>
</tbody>
</table>

4. Instruction Set Specification

4.2 Address Operation Instruction

Syntax

```plaintext
_setad (dst, im21, type)  // Sets the address register to immediate
_movad (dst, src, type)  // Copies the address register
```

dst Specify the destination address register with one of symbols below.

_A0_, _A1_, _A2_, _ZR_, _XR_, _YR_, and _DR_

im21 Specify up to 21 bit of byte address.

type Specify the address attribute of destination register with one of symbols below.

_long, _int, _short, and _uchar (unsigned char)

src Specify the source address register and displacement d with one of symbols below.

_A0(d), _A1(d), _A2(d), _ZR(d), _XR(d), _YR(d), _DR(d), and _WA(d)

The value in the source address register added by the displacement is put into the destination register. 

d is a signed 18bit integer. However, the displacement d to be added actually is, as listed in Table 4.2, 
right-shifted according to the address attribute specified by type.

Table 4.2 Relationship between Address Attribute Specified by type and Displacement

<table>
<thead>
<tr>
<th>Address attribute type</th>
<th>Displacement to be added actually</th>
</tr>
</thead>
<tbody>
<tr>
<td>_long</td>
<td>d &lt;&lt; 2</td>
</tr>
<tr>
<td>_int</td>
<td>d &lt;&lt; 2 or d &lt;&lt; 1</td>
</tr>
<tr>
<td>_short</td>
<td>d &lt;&lt; 1</td>
</tr>
<tr>
<td>_uchar (unsigned char)</td>
<td>d</td>
</tr>
</tbody>
</table>

Example

```plaintext
fsa(pFsaReg, _setad (_A0_, 0x4000, _long));  // Set A0 to 0x4000 and set long type for the A0 address attribute
fsa(pFsaReg, _movad (_DR_, _A0(4), _short));
  // Set DR to A0 added by 8 and set short type for the DR address attribute
```

Example

```plaintext
extern const long alConst[];

  // Specify the constant array through the _FSA_GLOBAL_ macro function.
  fsa(pFsaReg, _setad (_YR_, _FSA_GLOBAL_(alConst), _long));
  // To set constant array to the address added by offset, add the byte offset on the outside of the _FSA_GLOBAL_ macro function.
  fsa(pFsaReg, _setad (_XR_, _FSA_GLOBAL_(alConst) + 20, _long));
  // fsa(pFsaReg, _setad (_XR_, _FSA_GLOBAL_(alConst + 5), _long)); // This is wrong
```
4. Instruction Set Specification

4.3 Data Operation Instruction

Syntax

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>( n, \text{dst}Z, \text{src}X, \text{src}Y )</td>
<td>[ \text{dst}Z = \text{src}X + \text{src}Y ]</td>
</tr>
<tr>
<td>SUB</td>
<td>( n, \text{dst}Z, \text{src}X, \text{src}Y )</td>
<td>[ \text{dst}Z = \text{src}X - \text{src}Y ]</td>
</tr>
<tr>
<td>MP</td>
<td>( n, \text{dst}Z, \text{src}X, \text{src}Y )</td>
<td>[ \text{dst}Z = \text{src}X \times \text{src}Y ]</td>
</tr>
<tr>
<td>MAC</td>
<td>( n, \text{dst}Z, \text{src}X, \text{src}Y )</td>
<td>[ \text{dst}Z = \text{ACR} + \text{src}X \times \text{src}Y ]</td>
</tr>
<tr>
<td>MMAC</td>
<td>( n, \text{dst}Z, \text{src}X, \text{src}Y )</td>
<td>[ \text{dst}Z = \text{ACR} - \text{src}X \times \text{src}Y ]</td>
</tr>
<tr>
<td>AND</td>
<td>( \text{dst}Z, \text{src}X, \text{src}Y )</td>
<td>[ \text{dst}Z = \text{src}X \land \text{src}Y ]</td>
</tr>
<tr>
<td>OR</td>
<td>( \text{dst}Z, \text{src}X, \text{src}Y )</td>
<td>[ \text{dst}Z = \text{src}X \lor \text{src}Y ]</td>
</tr>
</tbody>
</table>

- **n** Specify the repeat count. Specify a number between 1 and 8. If two or more is specified, be careful that the range of the displacement \( d \) for every memory operand is restricted between -16 and 15. (default range is between -31 and 31)

- **dstZ** Specify the destination with one of symbols below.
  - Memory operand: \_A0(d), \_A1(d), \_A2(d), \_ZR(d), and \_DR(d)
  - Register operand: \_XD\_, \_DR\_, \_AC\_, \_TZ\_, \_ST\_, and \_BS\_

- **srcX** Specify the X source with one of symbols below.
  - Memory operand: \_A0(d), \_A1(d), \_A2(d), \_XR(d), and \_DR(d)
  - Register operand: \_XD\_, \_DR\_, and \_TZ\_
  - Immediate operand: \_i0\_, \_i1\_

- **srcY** Specify the Y source with one of symbols below.
  - Memory operand: \_A0(d), \_A1(d), \_A2(d), \_ZR(d), \_XR(d), \_YR(d), and \_DR(d)
  - Register operand: \_XD\_, \_YD\_
  - Immediate operand: \_iM1\_, \_i1\_, \_i2\_, and \_i4\_

- **\_shift** Specify this option to perform a right-shift. However, since the right-shift is done at E3-stage, it is valid only for the destination (memory operands, \_TZ\_, \_XD\_) that goes through E3-stage.

- **\_round** Specify this option to add an offset for rounding. Valid only for the _MP instruction.

4.3.1 Memory Operand

For the data operation instruction, the memory operands can be specified that are designated by a circle in Table 4.3. When the memory operand is specified, the data is read/written with specified address register. (register indirect addressing)

Table 4.3 Memory Operand

<table>
<thead>
<tr>
<th>Register</th>
<th>Symbol</th>
<th>dstZ</th>
<th>srcX</th>
<th>srcY</th>
</tr>
</thead>
<tbody>
<tr>
<td>A0</td>
<td>A0(d)</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>A1</td>
<td>A1(d)</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>A2</td>
<td>A2(d)</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>ZR</td>
<td>ZR(d)</td>
<td>O</td>
<td>X</td>
<td>O</td>
</tr>
<tr>
<td>XR</td>
<td>XR(d)</td>
<td>X</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>YR</td>
<td>YR(d)</td>
<td>X</td>
<td>X</td>
<td>O</td>
</tr>
<tr>
<td>DR</td>
<td>DR(d)</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
</tbody>
</table>
4. Instruction Set Specification

4.3.2 Post Register Update

When the memory operand is specified, the address register can be updated at the same time when the data is operated by adding the displacement d. The value added actually to the address register depends on the address attribute (long, short, or uchar type) set up at the address register. (similar to the pointer of C language) Note that the default range of d is between -31 and 31. However, when the repeat count of instruction is two or more, the range is restricted to between -16 and 15.

<table>
<thead>
<tr>
<th>Address attribute</th>
<th>Displacement to be added actually</th>
</tr>
</thead>
<tbody>
<tr>
<td>long type</td>
<td>d &lt;&lt; 2</td>
</tr>
<tr>
<td>short type</td>
<td>d &lt;&lt; 1</td>
</tr>
<tr>
<td>uchar type (unsigned char)</td>
<td>d</td>
</tr>
</tbody>
</table>

4.3.3 Register Indirect Displacement

When _BR_ is specified as the displacement, the BRC0 register is available for the update of dstZ and srcX and the BRC1 register is available for the update of srcY. However, since both BRC0 and BRC1 are 5bit register, their range is between -16 and 15. Since these registers are also used for the bit-reverse addressing control, they are not used with the bit-reverse addressing in principle. The data of BRC0 and BRC1 can be changed by the _MOVBR_ instruction and can also be set by the host CPU.

4.3.4 Priority Order of Displacement

When the same address register are specified for dstZ, srcX, and srcY, the priority order of the address update is as follows:

$$dstZ < srcX < srcY$$

For example,

Example: fsa(pFsaReg, _ADD (1, _A0( -1), _A0(  1), _A0(  2)));

The displacement of _A0(  2) specified by srcY becomes valid in the code above and those of _A0( -1) and _A0(  1) specified by dstZ and srcX are ignored.

4.3.5 Size of Reading/Writing Data

The data type to be read/written is determined based on the address attribute of address register specified by the memory operand.
4. Instruction Set Specification

4.3.6 Register Operand

For the data operation instruction, the register operands can be specified that are designated by a circle in Table 4.5. For information on the arrangement of each register on the operation pipeline, refer to Figure 2.1 Operation Pipeline.

<table>
<thead>
<tr>
<th>Register</th>
<th>Symbol</th>
<th>dstZ</th>
<th>srcX</th>
<th>srcY</th>
</tr>
</thead>
<tbody>
<tr>
<td>XDR</td>
<td><em>XD</em></td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>YDR</td>
<td><em>YD</em></td>
<td>X</td>
<td>X</td>
<td>O</td>
</tr>
<tr>
<td>ACR</td>
<td><em>AC</em></td>
<td>O</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>DR</td>
<td><em>DR</em></td>
<td>O</td>
<td>O</td>
<td>X</td>
</tr>
<tr>
<td>TZ</td>
<td><em>TZ</em></td>
<td>O</td>
<td>O</td>
<td>X</td>
</tr>
<tr>
<td>ST</td>
<td><em>ST</em></td>
<td>O</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>ST</td>
<td><em>BS</em></td>
<td>O</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Table 4.5 Register Operand

When specifying a register operand for dstZ, be careful about the following items:

- The sign bit of operation result and lower 31bit is stored in DR. It is impossible to apply the arithmetic right-shift or saturation processing to the data stored in DR.
- In logical operation instruction (_AND, _OR instructions), the zero-extended data is stored in ACR regardless the sign bit of operation result.
- When TZ or XDR is set for destination, ACR is also overwritten. When the data is read from the X bus, XDR is overwritten.
- To execute the instruction to write the operation result into a register and then read the data correctly from the register in the subsequent instruction, one or two cycles of delay is required according to the register. For more information, refer to 4.12. Delay Slots Required for Avoiding Data Hazard.

4.3.7 Immediate Operand

For the data operation instruction, the immediates can be specified that are designated by a circle in Table 4.6.

<table>
<thead>
<tr>
<th>Immediate</th>
<th>Symbol</th>
<th>srcX</th>
<th>srcY</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td><em>iM1</em></td>
<td>X</td>
<td>O</td>
</tr>
<tr>
<td>0</td>
<td><em>i0</em></td>
<td>O</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td><em>i1</em></td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>2</td>
<td><em>i2</em></td>
<td>X</td>
<td>O</td>
</tr>
<tr>
<td>4</td>
<td><em>i4</em></td>
<td>X</td>
<td>O</td>
</tr>
</tbody>
</table>

Table 4.6 Immediate Operand

Example

fsa(pFsaReg, _MP (1, _DR_, _A0( 1), _iM1_));   // DR = *A0++ * -1;
fsa(pFsaReg, _SUB (1, _ZR( 0), _i0_, _YR( 1)));   // *ZR = 0 - *YR++;
fsa(pFsaReg, _AND (_ST_, _DR_, _i1_));   // Check whether DR's least significant bit is zero.
4. Instruction Set Specification

4.4 Right-Shift, Rounding, Saturation Process Control Instructions

**Syntax**

```
_SETSC (im5) [._DRs] [._re] [._sa1/_sa2/_sa3]
_MOVSC (src)    [._DRs] [._re] [._sa1/_sa2/_sa3]
```

- **im5**: Right-shift amount to be put in the SC register. The configurable range is between 0 and 63. If the value specified exceeds 32, the value subtracted by 32 becomes actual shift value.
- **src**: Specify the source register to be put in the SC register along with the displacement. The following six symbols can be specified. The symbols _CB(d) and _BR(d) represents CBC0 and BRC0 registers. The CBC1 and BRC1 registers cannot be specified.
  - _SC(d), _DR(d), _LP0(d), _LP1(d), _CB(d), and _BR(d)
- **_DRs**: When the _DRs option is specified, the right-shift, a data operation instruction, is carried out by using lower 6bit of the DR register. If this option is not specified, the right-shift uses the SC register. If lower 6bit of the DR register exceeds 32, the value subtracted by 32 becomes actual shift value.
- **_re**: When the _re option is specified, auto rounding is performed with the _round option in the _MP instruction. If not specified, auto rounding is disabled. (For details on auto rounding, see the section described later)
- **_sa1/_sa2/_sa3**: This option sets saturation processing. Select one of them. If nothing is set, the saturation processing becomes disabled. Table 4.7 shows each setting details.

<table>
<thead>
<tr>
<th>Setting symbol</th>
<th>Saturation processing</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>Saturation processing disabled</td>
</tr>
<tr>
<td>_sa1</td>
<td>Saturated to 8bit unsigned data (0 to 255)</td>
</tr>
<tr>
<td>_sa2</td>
<td>Saturated to 16bit signed data (-32768 to 32767)</td>
</tr>
<tr>
<td>_sa3</td>
<td>When dstZ is a memory operand, it depends on the address register attribute. When the address attribute is long type, the source is saturated to 32bit signed data. As for short and uchar types, saturated with _sa2 and _sa1 setting each. As for the register operand, the source is always saturated to 32bit signed data.</td>
</tr>
</tbody>
</table>

Note: When _sa3 is set, the source is saturated adaptively according to the data size to be stored.
4. Instruction Set Specification

4.4.1 Auto Rounding Process

The auto rounding function adds the rounded value according to right-shift amount of SC register. When the _SETSC instruction with the _re option is executed, auto rounding is allowed, and the _MP instruction with the _shift or _round option is executed as shown below, the rounded value is added to the multiplication result. (so-called 0.5 addition) Be careful that the rounded value is always generated from the right-shift amount of SC register even if the _SETSC or _MOVSC instruction with the _DRs option is executed. The rounded value is not generated from lower 6bit of the DR register.

```c
fsa(pFsaReg, _SETSC (22) _re); // Set SC to 22. Allow rounding process
fsa(pFsaReg, _MP (1, _AC_, _A0( 1), _YR( 1)) _round); // Perform ACR = *A0++ *YR++ + (1 << 21)
fsa(pFsaReg, _MOVY (1, _DR_, _i4_); // Perform DR = 4
fsa(pFsaReg, _MOVSC (_SC( 0)) _re _DRs); // Allow rounding process and change the operation to the shift with DR register
fsa(pFsaReg, _MP (1, _ZR( 1), _A0( 1), _YR( 1)) _shift); // Perform ACR = *A0++ *YR++ + (1 << 21); *ZR++ = ACR >> 4
...
```

---

**Example**

```c
fsa(pFsaReg, _SETSC (22));  // Set SC to 20. Disallow rounding process
fsa(pFsaReg, _MP (1, _ZR( 1), _A0( 1), _YR( 1)) _shift); // Perform ACR = *A0++ *YR++ + (1 << 17); *ZR = ACR >> 18;
```

---

<table>
<thead>
<tr>
<th>Shift value</th>
<th>Additional value</th>
<th>Shift value</th>
<th>Additional value</th>
<th>Shift value</th>
<th>Additional value</th>
<th>Shift value</th>
<th>Additional value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x000000000</td>
<td>0x000000800</td>
<td>0x000080000</td>
<td>24</td>
<td>0x080000000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0x000000001</td>
<td>0x000001000</td>
<td>0x000100000</td>
<td>25</td>
<td>0x010000000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0x000000002</td>
<td>0x000002000</td>
<td>0x000200000</td>
<td>26</td>
<td>0x020000000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0x000000004</td>
<td>0x000004000</td>
<td>0x000400000</td>
<td>27</td>
<td>0x040000000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0x000000008</td>
<td>0x000008000</td>
<td>0x000800000</td>
<td>28</td>
<td>0x080000000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0x000000010</td>
<td>0x000010000</td>
<td>0x001000000</td>
<td>29</td>
<td>0x100000000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>0x000000020</td>
<td>0x000020000</td>
<td>0x002000000</td>
<td>30</td>
<td>0x200000000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>0x000000040</td>
<td>0x000040000</td>
<td>0x004000000</td>
<td>31</td>
<td>0x400000000</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note: When the right-shift value exceeds 32, the value subtracted by 32 becomes additional value.
4. Instruction Set Specification

4.5 Control Instruction for Conditional Execution

Syntax

\_SETCD \( (condition) \)

**condition**
The following symbols can be specified. The specified condition allowing execution is put in the CD register. Execute this instruction to allow you to control instructions based on this allowing condition from the following instructions. When the conditional execution control becomes unnecessary, execute \_SETCD(_alw) to clear the control.

<table>
<thead>
<tr>
<th>condition</th>
<th>Condition allowing execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>_alw</td>
<td>Always allowed</td>
</tr>
<tr>
<td>_lez</td>
<td>Allowed when zero or less</td>
</tr>
<tr>
<td>_gez</td>
<td>Allowed when zero or more</td>
</tr>
<tr>
<td>_ltz</td>
<td>Allowed when less than zero</td>
</tr>
<tr>
<td>_gtz</td>
<td>Allowed when greater than zero</td>
</tr>
<tr>
<td>_eqz</td>
<td>Allowed when zero</td>
</tr>
<tr>
<td>_nez</td>
<td>Allowed when other than zero</td>
</tr>
<tr>
<td>_aln</td>
<td>Always not allowed</td>
</tr>
</tbody>
</table>

4.5.1 Execution Control for Conditional Instruction

When a data operation instruction is executed by specifying _ST_ or _BS_ for dstZ, the ST register is set to the data status by evaluating all bits of the operation result. The data status is managed based on three flags: less than zero, equal to zero, greater than zero. After comparing this status with the condition allowing execution in the CD register set by the \_SETCD instruction, the writing of destination in subsequent data operation instructions and break of program loops are controlled. (Caution: Other instructions than data operation instructions and the address register update of data operation instructions cannot be controlled conditionally.)

Example

// When the data of *A0 is negative, multiply it by -1.
fsa(pFsaReg, _SETCD (_ltz)); // If less than zero, turn ON the execution
fsa(pFsaReg, _MP (1, _ST_, _A0(0), _i1_)); // Judge whether *A0 is positive or negative and set ST to the result
fsa(pFsaReg, _MP (1, _A0(0), _XD_, _iM1_)); // If negative, multiply it by -1 and write the result into *A0. Since they are the same, specify _XD_.
fsa(pFsaReg, _SETCD (_alw)); // When conditional execution ends, return to normal execution. (Don't forget!)
4. Instruction Set Specification

4.5.2 Conditional Program Loop Control

To control conditional program loop, set \texttt{dstZ} for data operation instruction to \texttt{_BS_} to set the data status. When the program loops because of the \texttt{LOOP} instruction etc., compare the data status with the condition allowing execution. When the execution is disabled, cancel the loop. However, the loop section length, position of data status setup instruction, existence of weight cycle for loading program because of bus conflict, and orders may cause the loop to run several times even after the condition for exiting the loop is actually met. The conditional program loop control should be used in the program that does not suffer any impact from extra running of the loop.

Example

\begin{verbatim}
// When the data of *A0 is less than zero, set TZ to -1. When the data is equal to zero, set TZ to zero. When greater than zero, set TZ to one.
fsa(pFsaReg, _SUB (1, _ST_, _i0_, _A0(0)));  // Check the data status of *A0.
fsa(pFsaReg, _SETCD (_ltz));          // If less than zero, turn ON the execution
fsa(pFsaReg, _ADD (1, _TZ_, _i0_, _iM1_)); // When the data of *A0 is less than zero, set TZ to -1
fsa(pFsaReg, _SETCD (_eqz));          // If equal to zero, turn ON the execution
fsa(pFsaReg, _ADD (1, _TZ_, _i1_, _iM1_)); // When the data of *A0 is equal to zero, set TZ to zero
fsa(pFsaReg, _SETCD (_gtz));          // If greater than zero, turn ON the execution
fsa(pFsaReg, _ADD (1, _TZ_, _i0_, _i1_)); // When the data of *A0 is greater than zero, set TZ to one
fsa(pFsaReg, _SETCD (_alw));         // When conditional execution ends, return to normal execution. (Don't forget!)

Example

// Calculate the number of significant figures (bits) of the data of *A0 and store the result in TZ.
fsa(pFsaReg, _ADD (1, _DR_, _i0_, _iM1_));  // Set DR to -1
fsa(pFsaReg, _SETZR (1, _TZ_));            // Set TZ to zero
fsa(pFsaReg, _SETCD (_gtz));               // If greater than zero, turn ON the execution
fsa(pFsaReg, _LOOP (31) { 
  fsa(pFsaReg, _ADD (1, _BS_, _DR_, _A0(0)));  // Set ST to DR added by the data of *A0. If ST is less than zero, exit the loop.
  fsa(pFsaReg, _MP  (1, _DR_, _DR_, _i2_));  // If DR added by the data of *A0 is greater than or equal to zero,
  left-shift DR by one bit
  fsa(pFsaReg, _ADD (1, _TZ_, _TZ_, _i1_));  // If DR added by the data of *A0 is greater than or equal to zero, add one to TZ
}
fsa(pFsaReg, _SETCD (_alw));  // When conditional execution ends, return to normal execution. (Don't forget!)
\end{verbatim}

When a loop is multiple, conditional controllable program loop is the innermost loop only.
4. Instruction Set Specification

4.6 Program Loop Instruction

Syntax

\[ \text{\_LOOP}(\text{count}) \] // Immediate loop
\[ \text{\_LOOPx}(\text{lpcr}) \] // Register indirect loop

**count**  Specify the loop count with an immediate between 1 and 257. If one is specified, the same operation as the _NOP instruction applies.

**lpcr**  Specify the loop control register with following symbols. The numeric value stored in specified loop control register indicates the loop count. If the loop control register is zero, the same operation as the _NOP instruction applies. The loop control register can be set and changed with the _SETLP and _MOVLP instructions.

\[ \_LP0, \_LP1 \]

4.6.1 Specifying Loop Section

Specify the loop section by enclosing it with braces, as shown in the example below. The number of instructions in the loop section can be up to 256. If there are not more than two instructions in the loop section, the loop is performed by repeating the instructions in the fetch buffer. (on-fetch loop) Therefore, the instructions are not reloaded from the P bus in the loop.

**Example**

// Get the maximum value out of 64 pieces of data starting with *A0.
fsa(pFsaReg, _AND (_DR_, _i0_, _iM1_)); // Initialize DR to zero
fsa(pFsaReg, _SETCD (_ltz)); // If less than zero, turn ON the execution
fsa(pFsaReg, _LOOP (64)) {  // Loop two subsequent instructions 64 times.
  fsa(pFsaReg, _SUB (1, _ST_, _DR_, _A0( 1))); // Set STR to DR subtracted by *A0. If *A0 is greater, perform the following
  instruction
  fsa(pFsaReg, _ADD (1, _DR_, _i0_, _YD_));  // If *A0 is greater, copy it to DR.
  fsa(pFsaReg, _NOP); // In order to avoid data hazard, insert NOP
}
fsa(pFsaReg, _SETCD (_alw)); // When conditional execution ends, return to normal execution. (Don't forget!)

4.6.2 Nesting of Loop

In general, up to triple nesting is enabled. When the innermost loop is an on-fetch loop, up to fourfold loop is enabled. However, as shown in the example below, the loop instruction cannot be put in the on-fetch loop.

**Bad example**

fsa(pFsaReg, _LOOP (256)) {  // On-fetch loop since the scope of the loop is two instructions
  fsa(pFsaReg, _LOOP (256)) { // On-fetch loop also since the scope of the loop is an instruction
    fsa(pFsaReg, _MOVX (1, _ZR( 1), _A0( 1)));
  }
}
4. Instruction Set Specification

4.7 Operation Instruction Using Loop Control Register

Syntax

```markdown
_SETLP (dst, im8)
_MOVLP (dst, src)
```

**dst** Specify the destination loop control register with one of symbols below.

- _LP0_, _LP1_

**im8** Specify a number in the range of 0 to 255.

**src** Specify the source register to be copied along with the displacement. The following six symbols can be specified.

The symbols _CB(d) and _BR(d) represents CBC0 and BRC0 registers. The CBC1 and BRC1 registers cannot be specified.

- _LP0(d), _LP1(d), _DR(d), _SC(d), _CB(d), and _BR(d)

Example

```c
// Example of irregular product-sum operation. The _MAC instruction carries out the product-sum operation once only at first. // Then, the product-sum count increases by one each time the operation is carried out.
fsa(pFsaReg, _SETLP (_LP0_, 0)); // Set LPC0 to zero
fsa(pFsaReg, _AND (_AC_, _i0_, _iM1_)); // Initialize ACR to zero
fsa(pFsaReg, _LOOP (32)) { // Loop four subsequent instructions 32 times.
    fsa(pFsaReg, _MOVAD (_XR_, _A0( 0), _long)); // Copy A0 to XR (initialize XR)
    fsa(pFsaReg, _LOOPx (_LP0_)) { // Loop a subsequent instruction as many times as specified by LPC0.
        fsa(pFsaReg, _MAC (1, _AC_, _XR( 1), _YR( 1))); // Perform the product-sum operation of ACR += *XR++ * *YR++. 
    } // Add one to LPC0. (Increase the loop count by one.)
}
```
4. Instruction Set Specification

4.8 Circular Buffer Addressing Control Instruction

Syntax

\[
\begin{align*}
 &_\text{SETCB} \quad (_\text{CB0}(\ flag0, \ param0), \ _\text{CB1}(\ flag1, \ param1)) \\
&_\text{MOVCB} \quad (_\text{CB0}(\ flag0, \ param0_{\text{disp}}), \ _\text{CB1}(\ flag1, \ param1_{\text{disp}}))
\end{align*}
\]

flag0, flag1 These flag bits set enable/disable of circular buffer addressing for each address register during the address generation. These settings are put in the CBREF0 and CBREF1 registers. Be careful that they are common to the enable/disable setting of bit-reverse addressing. (The symbols X and O in the table below represent disable and enable.)

Table 4.10 Circular Buffer Enable Flag

<table>
<thead>
<tr>
<th>flag0</th>
<th>A0</th>
<th>A1</th>
<th>A2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>X</td>
<td>O</td>
</tr>
<tr>
<td>2</td>
<td>X</td>
<td>O</td>
<td>X</td>
</tr>
<tr>
<td>3</td>
<td>X</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>4</td>
<td>O</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>5</td>
<td>O</td>
<td>X</td>
<td>O</td>
</tr>
<tr>
<td>6</td>
<td>O</td>
<td>O</td>
<td>X</td>
</tr>
<tr>
<td>7</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>flag1</td>
<td>ZR</td>
<td>XR</td>
<td>YR</td>
</tr>
<tr>
<td>-------</td>
<td>----</td>
<td>----</td>
<td>----</td>
</tr>
<tr>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>X</td>
<td>O</td>
</tr>
<tr>
<td>2</td>
<td>X</td>
<td>O</td>
<td>X</td>
</tr>
<tr>
<td>3</td>
<td>X</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>4</td>
<td>O</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>5</td>
<td>O</td>
<td>X</td>
<td>O</td>
</tr>
<tr>
<td>6</td>
<td>O</td>
<td>O</td>
<td>X</td>
</tr>
<tr>
<td>7</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
</tbody>
</table>

param0, param1 Specifies the size of circular buffers. param0 is common to A0, A1, and A2 and is stored in the CBC0 register. param1 is common to ZR, XR, and YR and is stored in the CBC1 register. For details, see 4.8.1 Specifying Size of Circular Buffers.

param0_{disp}, param1_{disp} Displacement to be added to the size setting of circular buffers. The value can be between -32 and 31. param0_{disp} is the displacement of CBC0 and param1_{disp} is that of CBC1.

4.8.1 Specifying Size of Circular Buffers

Specify the size of circular buffers with param0 and param1 for the _SETCB instruction. Even if the setting value is the same, be careful that the buffer size may vary according to the address attribute of each register. The starting address of circular buffer should be at an alignment position according to the buffer size to be configured. For example, when the buffer size is 256 byte, the lower 8 bit of starting address should be zero.

Table 4.11 Specifying Circular Buffer Size

<table>
<thead>
<tr>
<th>param0/param1</th>
<th>uchar type</th>
<th>short type</th>
<th>long type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>8</td>
<td>16</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>16</td>
<td>32</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td>32</td>
<td>64</td>
</tr>
<tr>
<td>5</td>
<td>32</td>
<td>64</td>
<td>128</td>
</tr>
<tr>
<td>6</td>
<td>64</td>
<td>128</td>
<td>256</td>
</tr>
<tr>
<td>7</td>
<td>128</td>
<td>256</td>
<td>512</td>
</tr>
<tr>
<td>8</td>
<td>256</td>
<td>512</td>
<td>1024</td>
</tr>
<tr>
<td>9</td>
<td>512</td>
<td>1024</td>
<td>2048</td>
</tr>
<tr>
<td>10</td>
<td>1024</td>
<td>2048</td>
<td>4096</td>
</tr>
<tr>
<td>11</td>
<td>2048</td>
<td>4096</td>
<td>8192</td>
</tr>
<tr>
<td>12</td>
<td>4096</td>
<td>8192</td>
<td>16384</td>
</tr>
<tr>
<td>13</td>
<td>8192</td>
<td>16384</td>
<td>32768</td>
</tr>
<tr>
<td>14</td>
<td>16384</td>
<td>32768</td>
<td>65536</td>
</tr>
<tr>
<td>15</td>
<td>32768</td>
<td>65536</td>
<td>131072</td>
</tr>
</tbody>
</table>
4.9 Bit-Reverse Addressing Control Instruction

Syntax

\[
\begin{align*}
\_SETBR & (\_BR0(\text{flag0, param0}), \_BR1(\text{flag1, param1})) \\
\_MOVBR & (\_BR0(\text{flag0, param0\_disp}), \_BR1(\text{flag1, param1\_disp}))
\end{align*}
\]

**flag0, flag1** These flag bits set enable/disable of bit-reverse addressing for each address register during the address generation. These settings are put in the CBREF0 and CBREF1 registers. Be careful that they are common to the enable/disable setting of circular buffer addressing. (The symbols X and O in the table below represent disable and enable.)

Table 4.12 Bit-Reverse Enable Flag

<table>
<thead>
<tr>
<th>flag0</th>
<th>A0</th>
<th>A1</th>
<th>A2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>X</td>
<td>O</td>
</tr>
<tr>
<td>2</td>
<td>X</td>
<td>O</td>
<td>X</td>
</tr>
<tr>
<td>3</td>
<td>X</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>4</td>
<td>O</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>5</td>
<td>O</td>
<td>X</td>
<td>O</td>
</tr>
<tr>
<td>6</td>
<td>O</td>
<td>O</td>
<td>X</td>
</tr>
<tr>
<td>7</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>flag1</th>
<th>ZR</th>
<th>XR</th>
<th>YR</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>X</td>
<td>O</td>
</tr>
<tr>
<td>2</td>
<td>X</td>
<td>O</td>
<td>X</td>
</tr>
<tr>
<td>3</td>
<td>X</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>4</td>
<td>O</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>5</td>
<td>O</td>
<td>X</td>
<td>O</td>
</tr>
<tr>
<td>6</td>
<td>O</td>
<td>O</td>
<td>X</td>
</tr>
<tr>
<td>7</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
</tbody>
</table>

**param0, param1** Mode and range of the bit-reverse addressing. param0 is common to A0, A1, and A2 and is stored in the BRC0 register. param1 is common to ZR, XR, and YR and is stored in the BRC1 register. For more information, refer to 4.8.1 Specifying Size of Circular Buffers.

**param0\_disp, param1\_disp** Displacement to be added to the bit-reverse addressing setting. The value can be between -32 and 31. param0\_disp is the displacement of BRC0 and param1\_disp is that of BRC1.

4.9.1 Specifying Bit-Reverse Addressing

Specify the mode and range of the bit-reverse addressing with param0 and param1 for the _SETBR instruction. Reverse mode when param0 and param1 are less than 16 and rotation mode when not less than 16. The bit scramble of the rotation mode is determined in combination with the circular buffer setting value. The bit-reverse is done on a two pieces of data basis. So, the target address bit position for the bit-reverse is shifted according to the address attribute.
4. Instruction Set Specification

4.9.2 Reverse Mode

Figure 4.2 shows general bit-reverse addressing mode. The range of address bits to be reversed is determined by `param0` and `param1`, as listed in Table 4.13.

![Figure 4.2 Bit Sort in Reverse Mode](image)

<table>
<thead>
<tr>
<th><code>param0/param1</code></th>
<th>Range of bits to be reversed</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Least significant</td>
</tr>
<tr>
<td>0</td>
<td>No exchanged</td>
</tr>
<tr>
<td>1</td>
<td>S</td>
</tr>
<tr>
<td>2</td>
<td>S</td>
</tr>
<tr>
<td>3</td>
<td>S</td>
</tr>
<tr>
<td>4</td>
<td>S</td>
</tr>
<tr>
<td>5</td>
<td>S</td>
</tr>
<tr>
<td>6</td>
<td>S</td>
</tr>
<tr>
<td>7</td>
<td>S</td>
</tr>
<tr>
<td>8</td>
<td>S</td>
</tr>
<tr>
<td>9</td>
<td>S</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
</tbody>
</table>

Table 4.13 shows that S is determined by the address attribute set for the address register.

<table>
<thead>
<tr>
<th>Address attribute</th>
<th>Position of S</th>
</tr>
</thead>
<tbody>
<tr>
<td>long type</td>
<td>3</td>
</tr>
<tr>
<td>short type</td>
<td>2</td>
</tr>
<tr>
<td>uchar type</td>
<td>1</td>
</tr>
</tbody>
</table>
4. Instruction Set Specification

4.9.3 Rotation Mode

This mode moves as many upper bits as specified down. The range of address bits to be rotated is determined by param0 and param1, as listed in Table 4.15. The number of upper bits to be exchanged with lower bits is determined by the combination of param0/param1 and setting value of circular buffer, as listed in Table 4.16.

![Figure 4.3 Bit Sort in Rotation Mode](image)

Table 4.15 Range of Address Bits to Be Rotated

<table>
<thead>
<tr>
<th>param0/param1</th>
<th>Range of bits to be rotated</th>
<th>Least significant</th>
<th>Most significant</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Least significant</td>
<td>Most significant</td>
</tr>
<tr>
<td>16</td>
<td></td>
<td>No exchanged</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>S</td>
<td>S+1</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>S</td>
<td>S+2</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>S</td>
<td>S+3</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>S</td>
<td>S+4</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>S</td>
<td>S+5</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>S</td>
<td>S+6</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>S</td>
<td>S+7</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>S</td>
<td>S+8</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>S</td>
<td>S+9</td>
<td></td>
</tr>
</tbody>
</table>

Note: When RSVD is specified, the bit-reverse is not done.

Table 4.16 Number of Upper Bits to Be Moved

<table>
<thead>
<tr>
<th>param0/param1</th>
<th>Setting value of circular buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0-2</td>
</tr>
<tr>
<td>16</td>
<td>RSVD</td>
</tr>
<tr>
<td>17</td>
<td>1</td>
</tr>
<tr>
<td>18</td>
<td>RSVD</td>
</tr>
<tr>
<td>19</td>
<td>RSVD</td>
</tr>
<tr>
<td>20</td>
<td>RSVD</td>
</tr>
<tr>
<td>21</td>
<td>RSVD</td>
</tr>
<tr>
<td>22</td>
<td>RSVD</td>
</tr>
<tr>
<td>23</td>
<td>RSVD</td>
</tr>
<tr>
<td>24</td>
<td>RSVD</td>
</tr>
<tr>
<td>25</td>
<td>RSVD</td>
</tr>
</tbody>
</table>
4. Instruction Set Specification

4.10 Special Instruction

Syntax

<table>
<thead>
<tr>
<th>_NOP</th>
<th>[wait] [ie0] [ie1]</th>
<th>// No operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>_EXIT</td>
<td>[wait] [ie0] [ie1]</td>
<td>// Terminate the program</td>
</tr>
</tbody>
</table>

(wait) The _wait option stops executing instructions until preceding instruction finishes writing memory. Set this option along with interrupt notice when the host CPU is going to refer immediately to the result written into the memory. When passing the operation results by register using TZ register, etc., this option is not required.

(ie0) The _ie0 option notifies the host CPU of an interrupt by the interrupt request signal 0.

(ie1) The _ie1 option notifies the host CPU of an interrupt by the interrupt request signal 1.

Example

```c
fsa(pFsaReg, _EXIT _wait _ie0); // Terminate the program. The interrupt request signal 0 causes an interrupt notice.
fsa(pFsaReg, _NOP _ie0);        // Send an interrupt notice without terminating the program.
```

Syntax

<table>
<thead>
<tr>
<th>_BREAK</th>
<th>// Break instruction</th>
</tr>
</thead>
</table>

The _BREAK instruction cancels a program for debugging. This is equivalent to the _EXIT instruction with _wait and _ie1 options. The _BREAK instruction is available for use only in the debug mode. The address of _BREAK can be obtained by the FSAGetPC function in the debug mode.
4. Instruction Set Specification

4.11 Directive

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>_MOVX (n, dstZ, srcX) [shift]</td>
<td>// dstZ = srcX</td>
</tr>
<tr>
<td>_MOVY (n, dstZ, srcY) [shift]</td>
<td>// dstZ = srcY</td>
</tr>
<tr>
<td>_MOVAC (dstZ) [shift]</td>
<td>// dstZ = ACR</td>
</tr>
<tr>
<td>_SETZR (n, dstZ)</td>
<td>// dstZ = 0</td>
</tr>
</tbody>
</table>

**n** Specify the repeat count. Specify a number between 1 and 8. If two or more is specified, be careful that the range of the displacement d for every memory operand is restricted between -16 and 15. (default range is between -31 and 31)

**dstZ** Specify the destination with one of symbols below.

- Memory operand:  _A0(d)_ , _A1(d)_ , _A2(d)_ , _ZR(d)_ , and _DR(d)_
- Register operand: _XD_ , _DR_ , _AC_ , _TZ_ , _ST_ , and _BS_

**srcX** Specify the X source with one of symbols below.

- Memory operand:  _A0(d)_ , _A1(d)_ , _A2(d)_ , _XR(d)_ , and _DR(d)_
- Register operand: _XD_ , _DR_ , and _TZ_
- Immediate operand: _i0_ , _i1_

**srcY** Specify the Y source with one of symbols below.

- Memory operand:  _A0(d)_ , _A1(d)_ , _A2(d)_ , _ZR(d)_ , _XR(d)_ , _YR(d)_ , and _DR(d)_
- Register operand: _XD_ , _YD_
- Immediate operand: _iM1_ , _i1_ , _i2_ , and _i4_

**shift** Specify this option to perform a right-shift. However, since the right-shift is done at E3-stage, it is valid only for the destination (memory operands, _TZ_ , _XD_) that goes through E3-stage.
4. Instruction Set Specification

4.12 Delay Slots Required for Avoiding Data Hazard

Table 4.17 lists the number of delay slots required when subsequent instruction reads the destination operand of preceding instruction with source option. For example, when the number of delay slots is four, it is necessary to insert four nondependent instructions (or _NOP instruction, if there is no appropriate instruction) between preceding and succeeding instructions.

Table 4.17 Delays Slots Required for Avoiding Data Hazard

<table>
<thead>
<tr>
<th>Destination operand</th>
<th>Preceding instruction (write)</th>
<th>Succeeding instruction (read)</th>
<th>Delay slot</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory</td>
<td>Data operation instruction</td>
<td>Data operation instruction</td>
<td>4</td>
</tr>
<tr>
<td><em>DR</em></td>
<td>Data operation instruction</td>
<td>Data operation instruction</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Data operation instruction</td>
<td>Data operation instruction</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>_MOVAD, _MOVSC, _MOVLP</td>
<td>Data operation instruction</td>
<td>4</td>
</tr>
<tr>
<td>_SETAD, _MOVAD</td>
<td>Data operation instruction</td>
<td>Data operation instruction</td>
<td>0</td>
</tr>
<tr>
<td>_SETAD, _MOVAD</td>
<td>Data operation instruction</td>
<td>Data operation instruction</td>
<td>0</td>
</tr>
<tr>
<td>_SETAD, _MOVAD</td>
<td>_MOVAD, _MOVSC, _MOVLP</td>
<td>Data operation instruction</td>
<td>0</td>
</tr>
<tr>
<td><em>XD</em></td>
<td>Data operation instruction</td>
<td>Data operation instruction</td>
<td>2</td>
</tr>
<tr>
<td><em>TZ</em></td>
<td>Data operation instruction</td>
<td>Data operation instruction</td>
<td>2</td>
</tr>
<tr>
<td><em>SC</em></td>
<td>_SETSC, _MOVSC</td>
<td>Data operation instruction</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>_MOVAD, _MOVSC, _MOVLP</td>
<td>Data operation instruction</td>
<td>0</td>
</tr>
<tr>
<td><em>A0</em>, <em>A1</em>, <em>A2</em>, <em>ZR</em>, <em>XR</em>, <em>YR</em></td>
<td>_SETAD, _MOVAD</td>
<td>Data operation instruction</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>_SETAD, _MOVAD</td>
<td>_SETAD, _MOVAD</td>
<td>0</td>
</tr>
<tr>
<td><em>LP0</em>, <em>LP1</em></td>
<td>_SETLP, _MOVLP</td>
<td>_LOOPx</td>
<td>0</td>
</tr>
</tbody>
</table>

Note: The data operation instruction is seven: _ADD, _SUB, _MP, _MAC, _MMAC, _AND, and _OR.

When the data operation instruction uses the DR register as source operand, the DR register cannot be updated by the address operation instruction (_SETAD, _MOVAD) in two subsequent slots.

Moreover, when using the DR register as right-shift source for the data operation instruction (DRs option is specified in the _SETSC, _MOVSC instructions), the DR register cannot be updated by the address operation instruction (_SETAD, _MOVAD) in four subsequent slots.

Table 4.18 Passing Hazard Occurred in DR Register

<table>
<thead>
<tr>
<th>Source operand</th>
<th>Preceding instruction (read)</th>
<th>Succeeding instruction (write)</th>
<th>Delay slot</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>DR</em></td>
<td>Data operation instruction</td>
<td>_SETAD, _MOVAD</td>
<td>2</td>
</tr>
<tr>
<td><em>DR</em> (right-shift source)</td>
<td>Data operation instruction</td>
<td>_SETAD, _MOVAD</td>
<td>4</td>
</tr>
</tbody>
</table>
5. Register Specification and Low Level API Function

This chapter describes the FSA's register specification and the specification of low level API functions for the host CPU to access these registers.

5.1 Register Map

Table 5.1  Register Map

<table>
<thead>
<tr>
<th>Register Symbol</th>
<th>Register Name</th>
<th>Low-level API function</th>
<th>Host CPU Access</th>
<th>Offset Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTR</td>
<td>Control Register</td>
<td>FSASetCTR, FSAGetCTR, FSAGetMode</td>
<td>R/W</td>
<td>0x0000</td>
</tr>
<tr>
<td>PC</td>
<td>Program Counter</td>
<td>FSASetPC, FSAGetPC</td>
<td>R/W</td>
<td>0x0004</td>
</tr>
<tr>
<td>A0</td>
<td>Address Register A0</td>
<td>FSASetA0, FSAGetA0</td>
<td>R/W</td>
<td>0x0008</td>
</tr>
<tr>
<td>A1</td>
<td>Address Register A1</td>
<td>FSASetA1, FSAGetA1</td>
<td>R/W</td>
<td>0x000C</td>
</tr>
<tr>
<td>A2</td>
<td>Address Register A2</td>
<td>FSASetA2, FSAGetA2</td>
<td>R/W</td>
<td>0x0010</td>
</tr>
<tr>
<td>ZR</td>
<td>Address Register ZR</td>
<td>FSASetZR, FSAGetZR</td>
<td>R/W</td>
<td>0x0014</td>
</tr>
<tr>
<td>XR</td>
<td>Address Register XR</td>
<td>FSASetXR, FSAGetXR</td>
<td>R/W</td>
<td>0x0018</td>
</tr>
<tr>
<td>YR</td>
<td>Address Register YR</td>
<td>FSASetYR, FSAGetYR</td>
<td>R/W</td>
<td>0x001C</td>
</tr>
<tr>
<td>DR</td>
<td>Address/Data Register DR</td>
<td>FSASetDR, FSAGetDR, FSAGetDRDat, FSAGetDRDat</td>
<td>R/W</td>
<td>0x0020</td>
</tr>
<tr>
<td>WA</td>
<td>Work Address Register</td>
<td>FSAGetWORK</td>
<td>R</td>
<td>0x0024</td>
</tr>
<tr>
<td>CBCR</td>
<td>Circular Buffer Control Register</td>
<td>FSASetCBCR, FSAGetCBCR</td>
<td>R/W</td>
<td>0x0028</td>
</tr>
<tr>
<td>BRCR</td>
<td>Bit-reverse Control Register</td>
<td>FSASetBRCR, FSAGetBRCR</td>
<td>R/W</td>
<td>0x002C</td>
</tr>
<tr>
<td>LPCR</td>
<td>Loop Control Register</td>
<td>FSASetLPCR, FSAGetLPCR</td>
<td>R/W</td>
<td>0x0030</td>
</tr>
<tr>
<td>OPPR</td>
<td>Operation Control Register</td>
<td>FSASetOPPR, FSAGetOPPR</td>
<td>R/W</td>
<td>0x0034</td>
</tr>
<tr>
<td>TZ</td>
<td>Data Register TZ</td>
<td>FSASetTZ, FSAGetTZ</td>
<td>R/W</td>
<td>0x0038</td>
</tr>
<tr>
<td>VerInfo</td>
<td>Version Information Register</td>
<td>FSAGetVerInfo</td>
<td>R</td>
<td>0x003C</td>
</tr>
<tr>
<td>BAR</td>
<td>Break Address Register</td>
<td>FSASetBAR, FSAGetBAR</td>
<td>R/W</td>
<td>0x0040</td>
</tr>
<tr>
<td>PC</td>
<td>Program Counter</td>
<td>FSASetOnlyPC</td>
<td>R/W</td>
<td>0x0044</td>
</tr>
<tr>
<td>IFR</td>
<td>Instruction Fetch Register</td>
<td>FSAGetIFR</td>
<td>R</td>
<td>0x0048</td>
</tr>
<tr>
<td>XOAR</td>
<td>X-bus Output Address Register</td>
<td>FSAGetXOAR</td>
<td>R</td>
<td>0x004C</td>
</tr>
<tr>
<td>YOAR</td>
<td>Y-bus Output Address Register</td>
<td>FSAGetYOAR</td>
<td>R</td>
<td>0x0050</td>
</tr>
<tr>
<td>ZOAR</td>
<td>Z-bus Output Address Register</td>
<td>FSAGetZOAR</td>
<td>R</td>
<td>0x0054</td>
</tr>
<tr>
<td>XDR</td>
<td>X-bus Data Register</td>
<td>FSAGetXDR</td>
<td>R</td>
<td>0x0058</td>
</tr>
<tr>
<td>YDR</td>
<td>Y-bus Data Register</td>
<td>FSAGetYDR</td>
<td>R</td>
<td>0x005C</td>
</tr>
<tr>
<td>ZDR</td>
<td>Z-bus Data Register</td>
<td>FSAGetZDR</td>
<td>R</td>
<td>0x0060</td>
</tr>
<tr>
<td>ACR0</td>
<td>Accumulator Register 0</td>
<td>FSAGetACR0</td>
<td>R</td>
<td>0x0064</td>
</tr>
<tr>
<td>ACR1</td>
<td>Accumulator Register 1</td>
<td>FSAGetACR1</td>
<td>R</td>
<td>0x0068</td>
</tr>
<tr>
<td>ACR2</td>
<td>Accumulator Register 2</td>
<td>FSAGetACR2</td>
<td>R</td>
<td>0x006C</td>
</tr>
</tbody>
</table>
5. Register Specification and Low Level API Function

5.2 CTR Register

5.2.1 Register Specification

REG[0000h] Control Register
Default = 00000000h

<table>
<thead>
<tr>
<th>Bit</th>
<th>Name</th>
<th>Read/Write</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>Reserved</td>
<td>N/A</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>Error</td>
<td>N/A</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>IRQ0/1 Raw Status</td>
<td>N/A</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>IRQ0 Raw Status</td>
<td>N/A</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>IRQ1 Raw Status</td>
<td>N/A</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>RM[1:0] Run Mode</td>
<td>01</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Stop</td>
<td>N/A</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Run</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Stop</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Run</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Stop</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Run</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Stop</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Run</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Stop</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Run</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Bit 0  Run bit
       Reading this bit has following meaning:
       0: Ready status.
       1: Busy status.
       Writing this bit has following effect:
       0: No effect
       1: Start running program from the address that was set in PC.

Bit 1  Stop bit (Write Only)
       Writing this bit has following effect:
       0: No effect
       1: Stop running program.

Bit 2-3 RM[1:0] : Run mode bits
       Reading RM[1:0] has following meaning:
       00: Run mode is normal mode.
       01: Run mode is debug mode.
       Writing RM[1:0] has following effect:
       10: Run mode is set to normal mode.
       11: Run mode is set to debug mode.
       0X: No effect

Note:
1. RM[1] is write only. When 1 is written to RM[1], writing RM[0] becomes valid.
2. The read value of RM[1] is always 0.

Bit 4-5 IRQ0/1 Raw Status bit
       Reading this bit has following meaning:
       0: The corresponding interrupt request is not occurred.
       1: The corresponding interrupt request is occurred.
       Writing this bit has following effect:
       0: No effect
       1: Clear the corresponding interrupt request.

Bit 8  Reserved instruction error (Read Only)
       When reserved instruction error occurs, this bit becomes 1.

Bit 16 Reset bit
       When 1 is written to this bit, FSA is reset.

Note:
Once a reserved instruction error occurs, the IRQ1 signal becomes activated.
5. Register Specification and Low Level API Function

5.2.2 API function

**FSASetCTR**

format: `void FSASetCTR(FSAREG *pFsaReg, const unsigned long ulValue)`

Arguments:
- `pFsaReg` Pointer of FSAREG structure
- `ulValue` The combination (or) of the following values can be specified.

<table>
<thead>
<tr>
<th>Values</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>FSA_RESET</td>
<td>FSA is reset.</td>
</tr>
<tr>
<td>FSA_IRQ1_CLR</td>
<td>IRQ1 interrupt is cleared.</td>
</tr>
<tr>
<td>FSA_IRQ0_CLR</td>
<td>IRQ0 interrupt is cleared.</td>
</tr>
<tr>
<td>FSA_STOP</td>
<td>Program is stopped running.</td>
</tr>
<tr>
<td>FSA_RUN</td>
<td>Program is started running.</td>
</tr>
</tbody>
</table>

**FSAGetCTR**

format: `unsigned int FSAGetCTR(FSAREG *pFsaReg)`

Arguments:
- `pFsaReg` Pointer of FSAREG structure

Return value:
- Current status of CTR register is returned.

**FSASetMode**

format: `void FSASetMode(FSAREG *pFsaReg, unsigned long ulValue)`

Arguments:
- `pFsaReg` Pointer of FSAREG structure
- `ulValue` The following values can be specified.

<table>
<thead>
<tr>
<th>Values</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>FSA_DEBUG</td>
<td>Run mode is set to debug mode.</td>
</tr>
<tr>
<td>FSA_NORMAL</td>
<td>Run mode is set to normal mode.</td>
</tr>
</tbody>
</table>

**Note:**
This function must be called before FSASetPC is called.
## 5. Register Specification and Low Level API Function

### 5.3 PC Register

#### 5.3.1 Register Specification

<table>
<thead>
<tr>
<th>REG[0004h/0044h] Program Counter</th>
<th>Default = Configurable</th>
<th>Read/Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC[15:0]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PC[31:16]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Bit 0-31 PC[31:0] : Program Counter
   Set the start address of target program before the Run bit is set to 1.
   PC[1:0] is read only. The reading value is always 0.

**Note:**
1. After a target program address is written to PC, FSA begins to load instruction immediately. Therefore PC should be set at first before the other registers are set.
2. Two addresses of 0004h and 0044h are assigned for PC. The following registers and the internal program loop status are initialized at the same time once 0004h is written. But when writing to 0044h, the Initialization is not done.

OPPR, CBREF0, CBREF1

3. The default is a value that is defined by `define FSA_ADDR_SEGMENT in fsa_cfg.v. And the writable bits are limited up to a length that is defined by `define FSA_ADDR_BITS.

#### 5.3.2 API function

**FSASetPC**

format: void FSASetPC(FSAREG *pFsaReg, unsigned long *pul)

Arguments:
- pFsaReg: Pointer of FSAREG structure
- pul: Pointer of address to be set

**Note:**
1. After a target program address is written to PC, FSA begins to load instruction immediately. So this API-function should be called earliest.
2. Once FSASetPC is called, CBREF0, CBREF1 and OPPR register are initialized to 0. Therefore the initialization of these registers is not necessary in beginning of FSA program.

**FSASetOnlyPC**

format: void FSASetOnlyPC(FSAREG *pFsaReg, const unsigned long *pul)

Arguments:
- pFsaReg: Pointer of FSAREG structure
- pul: Pointer of address to be set

**Note:**
1. After a target program address is written to PC, FSA begins to load instruction immediately.
2. When writing to PC by this API, the initialization that FSASetPC does is not done. Ordinarily FSASetPC should be used.
5. Register Specification and Low Level API Function

### 5.4 A0, A1, A2, ZR, XR, YR Register

#### 5.4.1 Register Specification

<table>
<thead>
<tr>
<th>REG[0008h-001Ch] A0, A1, A2, ZR, XR, YR Register</th>
<th>Read/Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default = configurable</td>
<td></td>
</tr>
</tbody>
</table>

**Address[15:0]**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Address[31:0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

**Address[31:16]**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Address[31:0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td></td>
</tr>
<tr>
<td>29</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td></td>
</tr>
<tr>
<td>27</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
</tr>
</tbody>
</table>

**Note:**

1. When an address is written by host CPU, the address attribute of the register is initialized to "int" type.
2. The default is a value that is defined by `define FSA_ADDR_SEGMENT in fsa_cfg.v. And the writable bits are limited up to a length that is defined by `define FSA_ADDR_BITS.

#### 5.4.2 API function

**FSAGetPC**

format unsigned long* FSAGetPC(FSAREG *pFsaReg)

**Arguments**

- pFsaReg Pointer of FSAREG structure

**Return value**

Current program counter address is returned as a pointer variable of unsigned long type.

**Note:**

In debug mode, the return value indicates current breakpoint address.

**FSASetA0/A1/A2/ZR/XR/YR**

format void FSASetA0(FSAREG *pFsaReg, const void *p)

**Arguments**

- pFsaReg Pointer of FSAREG structure
- p Pointer of address to be set

**Note:**

1. When an address is written, the address attribute of the register is initialized to "int" type.

**FSAGetA0/A1/A2/ZR/XR/YR**

format char* FSAGetA0(FSAREG *pFsaReg)

**Arguments**

- pFsaReg Pointer of FSAREG structure

**Return value**

Current address is returned as a pointer variable of char type.
5. Register Specification and Low Level API Function

5.5 DR Register

5.5.1 Register Specification

REG[0020h] DR Register
Default = 00000000h

<table>
<thead>
<tr>
<th>Bit 15-10</th>
<th>Bit 9-0</th>
</tr>
</thead>
<tbody>
<tr>
<td>15-10</td>
<td>9-0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bit 31-16</th>
<th>Bit 15-2</th>
<th>Bit 11-0</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-26</td>
<td>25-21</td>
<td>20-16</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bit 31</th>
<th>Bit 29</th>
<th>Bit 27</th>
<th>Bit 26</th>
<th>Bit 24</th>
<th>Bit 23</th>
<th>Bit 22</th>
<th>Bit 21</th>
<th>Bit 20</th>
<th>Bit 19</th>
<th>Bit 18</th>
<th>Bit 17</th>
<th>Bit 16</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>30</td>
<td>29</td>
<td>28</td>
<td>27</td>
<td>26</td>
<td>25</td>
<td>24</td>
<td>23</td>
<td>22</td>
<td>21</td>
<td>20</td>
<td>19</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bit 15</th>
<th>Bit 14</th>
<th>Bit 13</th>
<th>Bit 12</th>
<th>Bit 11</th>
<th>Bit 10</th>
<th>Bit 9</th>
<th>Bit 8</th>
<th>Bit 7</th>
<th>Bit 6</th>
<th>Bit 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>14</td>
<td>13</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bit 4</th>
<th>Bit 3</th>
<th>Bit 2</th>
<th>Bit 1</th>
<th>Bit 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**Note:**
1. DR register can be used for both data and address.
2. When a data or address is written by host CPU, the address attribute of the register is initialized to "int" type.

5.5.2 API function

**FSASetDR**

format void FSASetDR(FSAREG *pFsaReg, const void *p)

Arguments
- pFsaReg Pointer of FSAREG structure
- p Pointer of address to be set

**Note:**
1. When an address is written, the address attribute of the register is initialized to "int" type.

**FSAGetDR**

format char* FSAGetDR(FSAREG *pFsaReg)

Arguments
- pFsaReg Pointer of FSAREG structure

Return value
Current address is returned as a pointer variable of char type.

**FSASetDRDat**

format void FSASetDRDat(FSAREG *pFsaReg, const long lData)

Arguments
- pFsaReg Pointer of FSAREG structure
- lData Data to be set

**Note:**
1. When writing not address but data to DR, this API should be used.
2. Even if data is written by this API, the address attribute is initialized to "int" type.

**FSAGetDRDat**

format long FSAGetDRDat(FSAREG *pFsaReg)

Arguments
- pFsaReg Pointer of FSAREG structure

Return value
Current data is returned as a variable of long type.

**Note:**
1. When reading not address but data from DR, this API should be used.
5. Register Specification and Low Level API Function

5.6 WA Register

5.6.1 Register Specification

<table>
<thead>
<tr>
<th>REG[0024h] Work Address Register</th>
<th>Read Only</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default = Configurable</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address[15:0]</th>
<th>N/A</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16</td>
<td></td>
</tr>
</tbody>
</table>

Bit 0-20 Address[20:0]

Note:
This register is read only. The value is determined by logic design in advance. It is a value that is defined by `define FSA_WORK_ADDR in fsa_cfg.v.

5.6.2 API function

FSAGetWORK

<table>
<thead>
<tr>
<th>format</th>
<th>char* FSAGetWORK(FSAREG *pFsaReg)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arguments</td>
<td>pFsaReg Pointer of FSAREG structure</td>
</tr>
<tr>
<td>Return value</td>
<td>The work address is returned as a pointer variable of char type.</td>
</tr>
</tbody>
</table>
5. Register Specification and Low Level API Function

5.7 CBCR Register

5.7.1 Register Specification

<table>
<thead>
<tr>
<th>REG[0028h] Circular Buffer Control Register</th>
<th>Default = 00000000h</th>
<th>Read/Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit 0-3 CBC0[3:0] : Circular Buffer Control 0</td>
<td>N/A</td>
<td></td>
</tr>
<tr>
<td>Bit 8-11 CBC1[3:0] : Circular Buffer Control 1</td>
<td>N/A</td>
<td></td>
</tr>
</tbody>
</table>

Bit 0-3  CBC0[3:0] : Circular Buffer Control 0
Bit 8-11 CBC1[3:0] : Circular Buffer Control 1

5.7.2 API function

**FSASetCBCR**

void FSASetCBCR(FSAREG *pFsaReg, const unsigned long ulValue)

Arguments:
- pFsaReg: Pointer of FSAREG structure
- ulValue: Value to be set

**FSAGetCBCR**

unsigned long FSAGetCBCR(FSAREG *pFsaReg)

Arguments:
- pFsaReg: Pointer of FSAREG structure

Return value:
Current value is returned as a variable of unsigned long type.
5. Register Specification and Low Level API Function

5.8 BRCR Register

5.8.1 Register Specification

<table>
<thead>
<tr>
<th>Bit 0-4</th>
<th>Bit-reverse Control 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit 8-12</td>
<td>Bit-reverse Control 1</td>
</tr>
</tbody>
</table>

5.8.2 API function

```c
void FSASetBRCR(FSAREG *pFsaReg, const unsigned long uValue)
```

Arguments:
- `pFsaReg`: Pointer of FSAREG structure
- `uValue`: Value to be set

```c
unsigned long FSAGetBRCR(FSAREG *pFsaReg)
```

Arguments:
- `pFsaReg`: Pointer of FSAREG structure

Return Value:
Current value is returned as a variable of unsigned long type.
5. Register Specification and Low Level API Function

5.9 LPCR Register

5.9.1 Register Specification

| Bit 0-7 | LPC0[7:0] | Loop Control 0 |
| Bit 8-15 | LPC1[7:0] | Loop Control 1 |

LPC0/LPC1 is referred by _LOOPx instruction and specifies repetition of loop. When LPC0/LPC1 is 0, _LOOPx dose not loop (equals to _NOP)

5.9.2 API function

**FSASetLPCR**

void FSASetLPCR(FSAREG *pFsaReg, const unsigned long ulValue)

**FSAGetLPCR**

unsigned long FSAGetLPCR(FSAREG *pFsaReg)
5. Register Specification and Low Level API Function

5.10 OPPR Register

5.10.1 Register Specification

<table>
<thead>
<tr>
<th>Bit 0-4</th>
<th>SC[5:0] : Shift Control</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>This bit-field indicates current arithmetic right shift amount. The shift amount is used by arithmetic instructions with _shift option. From 0 to 32 can be set.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bit 6</th>
<th>SS : Shift Select</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>This bit selects which shift-source should be used.</td>
</tr>
<tr>
<td></td>
<td>0: Shift Control [4:0] is used as a shift amount</td>
</tr>
<tr>
<td></td>
<td>1: Lower 5-bit of DR is used as a shift amount.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bit 7</th>
<th>RE : Auto Rounding Enable</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>This bit selects to enable auto-rounding or disable.</td>
</tr>
<tr>
<td></td>
<td>0: Disable</td>
</tr>
<tr>
<td></td>
<td>1: Enable</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bit 8-9</th>
<th>SA[1:0] : Saturation Control</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>This bit-field indicates current saturation setting. The following setting is default configuration.</td>
</tr>
<tr>
<td></td>
<td>00: No saturation</td>
</tr>
<tr>
<td></td>
<td>01: Unsigned 8-bit saturation</td>
</tr>
<tr>
<td></td>
<td>10: Signed 16-bit saturation</td>
</tr>
<tr>
<td></td>
<td>11: Adaptive saturation by size of stored data</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bit 16-18</th>
<th>CD[2:0] : Conditional Control</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>This bit-field indicates current condition setting to judge data status.</td>
</tr>
<tr>
<td></td>
<td>000: Always enable</td>
</tr>
<tr>
<td></td>
<td>001: Great than 0 or equal to 0</td>
</tr>
<tr>
<td></td>
<td>010: Not equal to 0</td>
</tr>
<tr>
<td></td>
<td>011: Great than 0</td>
</tr>
<tr>
<td></td>
<td>100: Little than 0 or equal to 0</td>
</tr>
<tr>
<td></td>
<td>101: Equal to 0</td>
</tr>
<tr>
<td></td>
<td>110: Little than 0</td>
</tr>
<tr>
<td></td>
<td>111: Always disable</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bit 19</th>
<th>ES : Conditional Execution Status (Read only)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>This bit indicates a current result that was judged by data status and condition control setting.</td>
</tr>
<tr>
<td></td>
<td>0: true (Enable)</td>
</tr>
<tr>
<td></td>
<td>1: false (Disable)</td>
</tr>
</tbody>
</table>

**Note:**

OPPR register is cleared to 0 once PC is written.
5. Register Specification and Low Level API Function

5.10.2 API function

```
FSASetOPPR
format void FSASetOPPR(FSAREG *pFsaReg, const unsigned long ulValue)
Arguments pFsaReg Pointer of FSAREG structure
ulValue Value to be set

FSAGetOPPR
format unsigned long FSAGetOPPR(FSAREG *pFsaReg)
Arguments pFsaReg Pointer of FSAREG structure
Return value Current value is returned as a variable of unsigned long type.
```

5.11 TZ Register

5.11.1 Register Specification

```
REG[0038h] TZ Register
Default = 00000000h

<table>
<thead>
<tr>
<th>Bit</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>0</td>
</tr>
<tr>
<td>30</td>
<td>0</td>
</tr>
<tr>
<td>29</td>
<td>0</td>
</tr>
<tr>
<td>28</td>
<td>0</td>
</tr>
<tr>
<td>27</td>
<td>0</td>
</tr>
<tr>
<td>26</td>
<td>0</td>
</tr>
<tr>
<td>25</td>
<td>0</td>
</tr>
<tr>
<td>24</td>
<td>0</td>
</tr>
<tr>
<td>23</td>
<td>0</td>
</tr>
<tr>
<td>22</td>
<td>0</td>
</tr>
<tr>
<td>21</td>
<td>0</td>
</tr>
<tr>
<td>20</td>
<td>0</td>
</tr>
<tr>
<td>19</td>
<td>0</td>
</tr>
<tr>
<td>18</td>
<td>0</td>
</tr>
<tr>
<td>17</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>0</td>
</tr>
<tr>
<td>15-0</td>
<td>0</td>
</tr>
</tbody>
</table>
```

Bit 0-31 TZ[31:0]

5.11.2 API function

```
FSASetTZ
format void FSASetTZ(FSAREG *pFsaReg, const long lData)
Arguments pFsaReg Pointer of FSAREG structure
lData Data to be set

FSAGetTZ
format long FSAGetTZ(FSAREG *pFsaReg)
Arguments pFsaReg Pointer of FSAREG structure
Return value Current data is returned as a variable of long type.
```
5. Register Specification and Low Level API Function

5.12 VerInfo Register

5.12.1 Register Specification

![VerInfo Register Diagram]

Bit 0-31 VerInfo[31:0]

**Note:**
1. The default is a value that is defined by `define FSA_VERSION_INFO` in `fsa_cfg.v`.

5.12.2 API function

```
FSAGetVerInfo

format       unsigned long FSAGetVerInfo(FSAREG *pFsaReg)

Arguments     pFsaReg  Pointer of FSAREG structure

Return value  The version information is returned.
```
5. Register Specification and Low Level API Function

5.13 BAR Register

5.13.1 Register Specification

Bit 0-31 BAR[31:0]
After an address to break program is set, when the program runs on debug mode, the program stops when an instruction of the address is fetched to IFR.
BAR[1:0] is read only. The reading value is always 0.

Note:
1. The default is a value that is defined by `define FSA_ADDR_SEGMENT in fsa_cfg.v.
   And the writable bits are limited up to a length that is defined by `define FSA_ADDR_BITS.

5.13.2 API function

```
void FSASetBAR(FSAREG *pFsaReg, const unsigned long *pul)
  pFsaReg       Pointer of FSAREG structure
  pul           Pointer of address to be set

unsigned long * FSAGetBAR(FSAREG *pFsaReg)
  pFsaReg       Pointer of FSAREG structure

Current break address is returned as a pointer variable of unsigned long type.
```
5. Register Specification and Low Level API Function

5.14 IFR Register

5.14.1 Register Specification

Bit 0-31 IFR [31:0]
IFR register indicates an instruction code to be executed next after program is stopped by a break address or _EXIT instruction.

5.14.2 API function

```c
FSAGetIFR

| format   | unsigned long FSAGetIFR(FSAREG *=pFsaReg) |
| Arguments| pFsaReg Pointer of FSAREG structure       |
| Return value | A current instruction code on IFR is returned.  |
```
5. Register Specification and Low Level API Function

5.15 XOAR Register

5.15.1 Register Specification

Bit 0-31 XOAR[31:0]
XOAR register indicates an address value that was read from X-bus at the last time.

Note:
1. The default is a value that is defined by `define FSA_ADDR_SEGMENT in fsa_cfg.v.

5.15.2 API function

FSAGetXOAR

format char * FSAGetXOAR(FSAREG **pFsaReg)
Arguments pFsaReg Pointer of FSAREG structure
Return value An address value that was read from X-bus at the last time is returned as a pointer variable of char type.
5. Register Specification and Low Level API Function

5.16 YOAR Register

5.16.1 Register Specification

<table>
<thead>
<tr>
<th>Bit 0-31</th>
<th>YOAR[31:0]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>YOAR register indicates an address value that was read from Y-bus at the last time.</td>
</tr>
</tbody>
</table>

**Note:**
1. The default is a value that is defined by `define FSA_ADDR_SEGMENT in fsa_cfg.v.`
5. Register Specification and Low Level API Function

5.17 ZOAR Register

5.17.1 Register Specification

Bit 0-31 ZOAR[31:0]
ZOAR register indicates an address value that was written to Z-bus at the last time.

Note:
1. The default is a value that is defined by `define FSA_ADDR_SEGMENT in fsa_cfg.v.

5.17.2 API function

```
FSAGetZOAR
format char "FSAGetZOAR(FSAREG =\"pFsareg\")"
Arguments pFsareg Pointer of FSAREG structure
Return value An address value that was written to Z-bus at the last time is returned as a pointer variable of char type.
```
5. Register Specification and Low Level API Function

5.18 XDR Register

5.18.1 Register Specification

Bit 0-31 XDR[31:0]
XDR register indicates a data that was read from X-bus at the last time.

5.18.2 API function

FSAGetXDR
format Arguments
long FSAGetXDR(PSAREG *pFsReg)
Return value
pFsReg Pointer of FSAREG structure
A data that was read from X-bus at the last time is returned as a variable of long type.
5. Register Specification and Low Level API Function

5.19 YDR Register

5.19.1 Register Specification

Bit 0-31 YDR[31:0]
YDR register indicates a data that was read from Y-bus at the last time.

5.19.2 API function

FSAGetYDR

Format: long FSAGetYDR(FSAREG *pFsReg)
Arguments: pFsReg Pointer of FSAREG structure
Return value: A data that was read from Y-bus at the last time is returned as a variable of long type.
5. Register Specification and Low Level API Function

5.20 ZDR Register

5.20.1 Register Specification

Bit 0-31  ZDR[31:0]
ZDR register indicates a data that was written to Z-bus at the last time.

5.20.2 API function

FSAGetZDR

<table>
<thead>
<tr>
<th>Format</th>
<th>Arguments</th>
<th>Return value</th>
</tr>
</thead>
<tbody>
<tr>
<td>long FSAGetZDR(FSAREG *pFsaReg)</td>
<td>*pFsaReg Pointer of FSAREG structure</td>
<td>A data that was written to Z-bus at the last time is returned as a variable of long type.</td>
</tr>
</tbody>
</table>
5. Register Specification and Low Level API Function

5.21 ACR Register

5.21.1 Register Specification

Note:
The bit-length is configurable. The default length is 72-bits like the above.

5.21.2 API function

FSAGetACR0

<table>
<thead>
<tr>
<th>format</th>
<th>long FSAGetACR0(FSAREG *pFsaReg)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arguments</td>
<td>*pFsaReg, Pointer of FSAREG structure</td>
</tr>
<tr>
<td>Return value</td>
<td>*ACR[31:0] is returned as a variable of long type.</td>
</tr>
</tbody>
</table>

FSAGetACR1

<table>
<thead>
<tr>
<th>format</th>
<th>long FSAGetACR1(FSAREG *pFsaReg)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arguments</td>
<td>*pFsaReg, Pointer of FSAREG structure</td>
</tr>
<tr>
<td>Return value</td>
<td>*ACR[63:32] is returned as a variable of long type.</td>
</tr>
</tbody>
</table>

FSAGetACR2

<table>
<thead>
<tr>
<th>format</th>
<th>long FSAGetACR2(FSAREG *pFsaReg)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arguments</td>
<td>*pFsaReg, Pointer of FSAREG structure</td>
</tr>
<tr>
<td>Return value</td>
<td>*ACR[95:64] is returned as a variable of long type.</td>
</tr>
</tbody>
</table>
## Revision History

<table>
<thead>
<tr>
<th>Rev. No.</th>
<th>Date</th>
<th>Page</th>
<th>Category</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rev 1.0</td>
<td>2015/01/09</td>
<td>All</td>
<td>new</td>
<td></td>
</tr>
</tbody>
</table>
AMERICA
EPSON ELECTRONICS AMERICA, INC.
214 Devcon Drive,
San Jose, CA 95112, USA
Phone: +1-800-228-3964 FAX: +1-408-922-0238

EUROPE
EPSON EUROPE ELECTRONICS GmbH
Riesstrasse 15, 80992 Munich,
GERMANY
Phone: +49-89-14005-0 FAX: +49-89-14005-110

ASIA
EPSON (CHINA) CO., LTD.
7F, Jinbao Bldg., No.89 Jinbao St.,
Dongcheng District,
Beijing 100005, CHINA
Phone: +86-10-8522-1199 FAX: +86-10-8522-1125

SHANGHAI BRANCH
7F, Block B, Hi-Tech Bldg., 900 Yishan Road,
Shanghai 200233, CHINA
Phone: +86-21-5423-5577 FAX: +86-21-5423-4677

SHENZHEN BRANCH
12F, Dawning Mansion, Keji South 12th Road,
Hi-Tech Park, Shenzhen 518057, CHINA
Phone: +86-755-2699-3829 FAX: +86-755-2699-3838

EPSON HONG KONG LTD.
Unit 715-723, 7/F Trade Square, 681 Cheung Sha Wan Road,
Kowloon, Hong Kong.
Phone: +852-2585-4600 FAX: +852-2827-4346

EPSON TAIWAN TECHNOLOGY & TRADING LTD.
14F, No. 7, Song Ren Road,
Taipei 110, TAIWAN
Phone: +886-2-8786-6688 FAX: +886-2-8786-6660

EPSON SINGAPORE PTE., LTD.
1 HarbourFront Place,
#03-02 HarbourFront Tower One, Singapore 098633
Phone: +65-6586-5500 FAX: +65-6271-3182

SEIKO EPSON CORP.
KOREA OFFICE
5F, KLI 63 Bldg., 60 Yoido-dong,
Youngdeungpo-Ku, Seoul 150-763, KOREA
Phone: +82-2-784-6027 FAX: +82-2-767-3677

SEIKO EPSON CORP.
MICRODEVICES OPERATIONS DIVISION
Device Sales & Marketing Department
421-8, Hino, Hino-shi, Tokyo 191-8501, JAPAN
Phone: +81-42-587-5814 FAX: +81-42-587-5117