advanced digital chips inc.

EISC Software Developer Guide

Extendable Instruction Set Computer

English version
2008.xx.xx
Advanced Digital Chips Inc.

EISC Software Developer Guide

Office
8th Floor, KookMin 1 Bldg.,
1009-5, Daechi-Dong, Gangnam-Gu, Seoul, 135-280, Korea.
Tel : +82-2-2107-5800
Fax : +82-2-571-4890
URL : http://www.adc.co.kr

Index

Figure

Table

Chapter 1
Introduction

In this chapter presents basic information of EISC (Extendable Instruction Set Computer) which is developed by Advanced Digital Chips Inc. In addition, this chapter contains program development methodology through using EISC compiler.

This chapter includes below.

1.1 EISC architecture

1.1.1 Overview

Compared to the general data processing processors, the embedded processors require low price and low power, In the embedded processors, the program is stored in the ROM. Thus, if we can reduce the program code size, we can also reduce the size of the ROM, which takes lots of spaces in the die. As a result, we can provide the embedded processors with much cheaper price.

An EISC processor, developed in ADChips Inc. (http://www.adc.co.kr), provides optimized ISA (Instruction Set Architecture) for embedded application. The ISA can reduce the program size and less a memory access. It is also able to expand a immediate constant using expand instruction called 'LERI'. These feathers are advantages of both CISC and RISC architecture.

EISC adopts both the simple hardware structure of the RISC and the advantages of the CISC architecture resulting in improved performance. With the reduction of the code size, the size of a program is reduced to about 60% compared to the RISC processor and about to 80% to the CISC processor. Thus, the EISC processor has advantage in the field where the code density.

1.1.2 Software Development Environment

In order to develop an application on EISC system, developers must establish EISC development environment.

- IDE (Integrated Development Environment)
  - EISC Studio: Total solution for EISC system, it can be used on Microsoft Windows.

- Tool Chain based on GNU: It can be used on Linux and Microsoft Windows (Cygwin)
  - GNU C/C++ compiler
  - GNU assembler and linker
  - GNU debugger (GDB), Insight GUI debugger
  - GNU instruction set simulator
- ESCAsim
  - EISC System Cycle Accurate Simulator

1.1.3 About this document

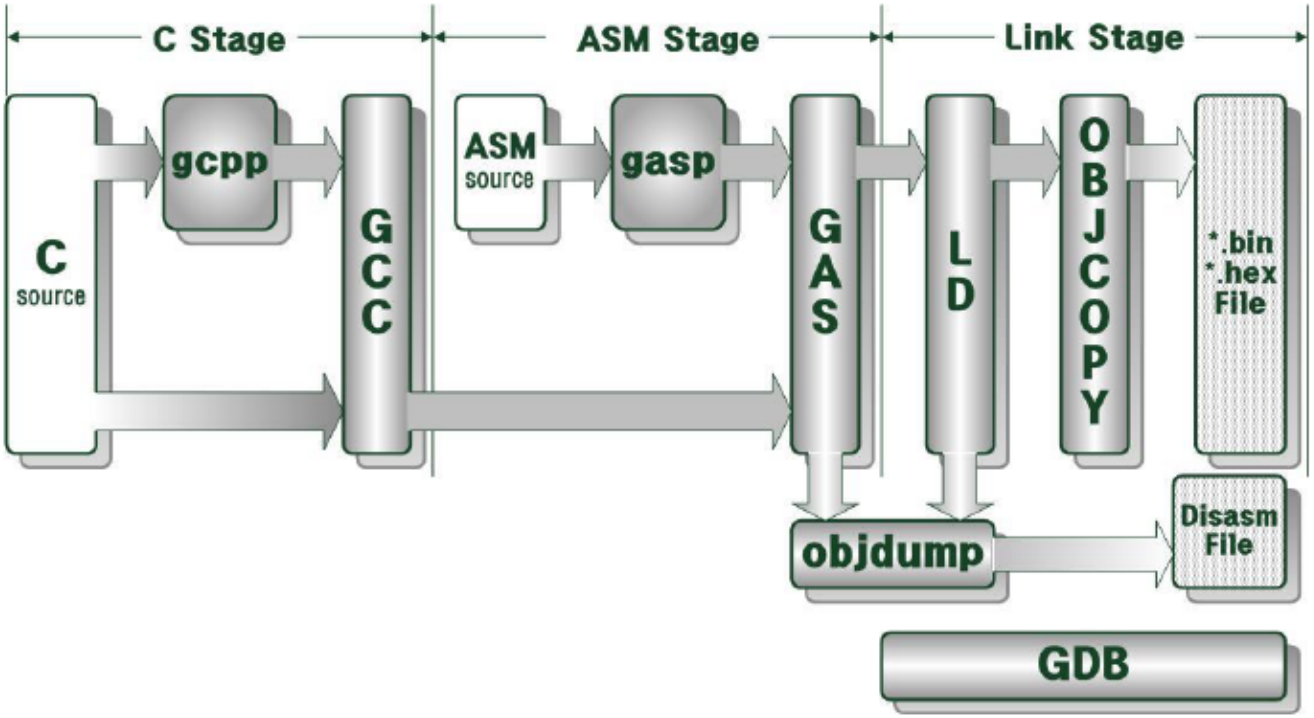
This document is written for those who develop an application on EISC system. A reader required basic knowledge of C/C++ language and assembler. In addition to the EISC processor is suitable for embedded system, it helps if you have a hardware details.

1.2 EISC Compiler Tool Chain

EISC compiler is a cross compiler based on GNU GCC. The GCC generates highly optimized code and has stability. You can find a GCC manual at http://gcc.gnu.org to get more information about GCC. The GCC manual documents how to use the GNU compiler, as well as their features.

Tool chain means a cross compile environment to development software on target system. It includes assembler, linker, C/C++ compiler, libraries and so on. These tools are called cross tools.

Cross tool chain generates an object file or an executable file for target system on host system. The host system indicates a system that operates compile, assembling and linking, and the target system can execute the output file from the cross tool chain. For example, if developer compiles a C source code on user PC and generate for EISC executable file, the PC is the host platform and the EISC is the target platform.



[Figure 1-1] A flow of C compiler

The [Figure. 1-1] shows a full flow of compilation. C compiler generates assembly code from C source code. After that, the assembler makes a object file as an output. Finally, the liker links the object files and libraries and generates an executable binary output.

ECOMI is EISC Compiler Installer that installs EISC compiler tool chain and Cygwin to

developer’s Microsoft windows system. User can install EISC compiler through executing a SETUP.EXE file. Also, if you already install the Cygwin emulator, you can install only the EISC tool chain. The EISC tool chain is installed to '\usr\local' where the Cygwin installed directory.

Directory	Description
bin	Executable files of EISC compiler tool cahin
lib	Contains libraries in order to make executable file such as libc, libm.
man	Manual for compiler tool chain.
include	Header files are located that needed by compiler
<target>-elf	Target specific directory.

1.3 Issues of General Programming

EISC processor is similar to RISC processor as a structural aspect. The EISC architecture supports several programming method in order to increase code density.

The EISC processor can access an aligned data, which are 4-byte word, 2-byte short and 1-byte character on memory. The data must allocate to memory with appropriate VMA. It means integer and long data type must have a VMA multiply by 4, and short data type must has VMA multiply by 2. A character data can be allocate everywhere on memory region.

The EISC compiler generates load / store instruction to access these data type with as the alignment.

1.3.1 Miss aligned data

Compiler does not allow data to locate at mis-aligned memory address even it shows higher performance and reduced code size. So the compiler assumes that data is aligned to memory. For example, the EISC compiler emits a LD instruction to read an integer type data from memory. If a VMA of the data is multiplied by 4, the LD operation works well. However, the VMA is not multiplied by 4(ex. 0x0006), the LD operation cannot be executed and Data Mis-align Error occurs.

Developer can read and write misaligned data with '__packed' attribute. Let's see a next C source code.

__attribute__((packed)) int *p;; // pointer to mis-aligned int

When compiler reads the value '*p', the EISC compiler does not use LD instruction. To get a valid data, compiler generates different codes such as shift operations and bit mask operations. In that case, obviously the code size will be increased, so developer had better not to use such the case.

1.3.2 Miss aligned field on structure data type

Similar to data alignment, members of structure are aligned to memory according to data type. Thus, compiler uses padding method between the members. In order to optimize memory size, sometimes developers use mis-aligned members. Compiler can access the mis-aligned data with several operations (shift and mask). This case also increases a code size but can reduce memory usage.

[Example code 1-1]

```
struct foo
{
        char a;
        int x;
};
```

[Example code 1-1] shows a structure data with 2 members one is character 'a' and another is integer 'x'. We can guess a memory alignment the structure easily. Both the two members 'a' and 'x'

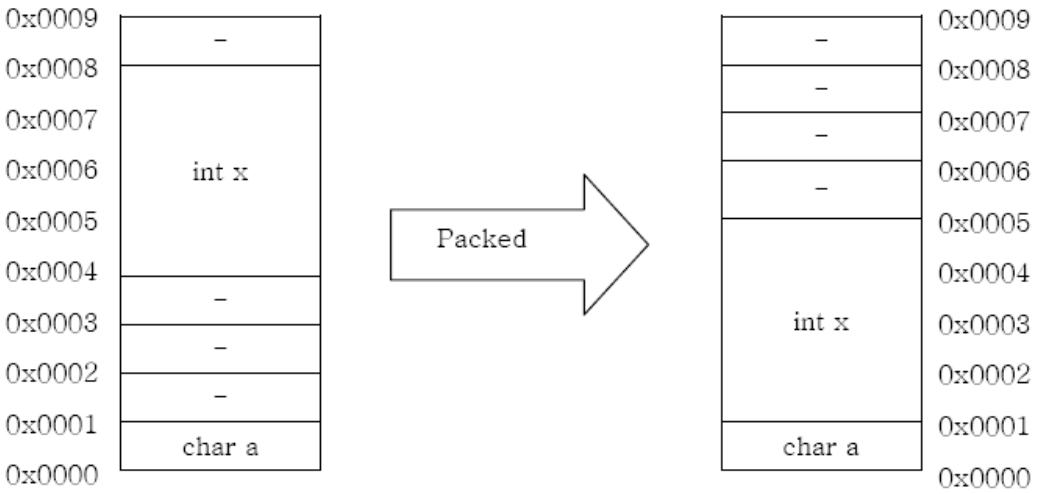
have 4-byte memory size and be aligned. The member 'a' has only 1-byte data size and spends 4-bytes on memory. An empty slot between 'a' and 'x' will be filled with trash value and it means 'padding'.

In order to remove the empty space, developer can use '__packed' attribute like as [Example code 1-2]

[Example code 1-2]

```
struct foo
{
        char a;
        int x __attribute__((packed));
};
```

The [Example code 1-2] shows a structure foo. A difference with [Example code 1-1] is that the member 'x' is packed. [Figure 1-2] shows the difference of memory alignment between [Example code 1-1] and [Example code 1-2].



[Figure 1-2] Two memories' alignment about the two example codes.

1.4 Developing Embedded S/W

1.4.1 Combination of C/C++ and assembly language

Individually compiled and assembled sources are be able to use together by linking. Also, by using inlining to C/C++, developers can use assembly language on C/C++ source. If you want to use the assembly language, you should refer to chapter 4.

1.4.2 Exceptions of processor

The EISC processor supports exceptions described below.

Exception	Description
RESET	To reset the EISC processor, signal send to processor from external
External Hardware Interrupt	External interrupt from an external hardware module. There are two kinds of EHI, one is autovectored interrupt and the other is vectored interrupt.
Software Interrupt(SWI)	Interrupt occurring from software
Non-Maskable Interrupt	Masking(denial of the interrupt request) is not allowed in the interrupt controller.
System Co-Processor Interrupt	Occurs during an access to memory
Co-porcessor Interrupt	Occurs during an access of the coprocessor
Breakpoint & Watchpoint Interrupt	An interrupt to support debugging
Bus Error & Double fault	If irrecoverable errors are detected in the instruction bus or data bus.
Undefined Instruction Exception	Occurs when try executing undefined instruction
Unimplemented Instruction Exception	An instruction is defined but not implemented yet.

Chapter 5 handles all of the exceptions detail.

Chapter 2
C/C++ Compiler & Binutils

This chapter explains usage of an EISC compiler and EISC binutils.

This chapter includes below.

2.1 EISC C/C++ Compiler

This section explains basic usage of the EISC compiler and compiler option. The EISC compiler is based on GCC (Gnu Compiler Collection) and a compiler options are almost same as GCC's. Compiler can accept one letter option like as '-o' and can accept multiple letters like as '-nostartfiles'. EISC software developer can make an executable file by using these options. In addition, '-v' option displays the EISC compiler's version information.

In this section, s/w developer can learn about compiling and linking method with the EISC compiler. Especially, this section includes explanation about bunch of options in order to make an application.

Generally, we can execute EISC compiler for compiling.
$ae32000-elf-gcc [Option] <Input_file>

This developer guide limits only for AE32000 processor. If you develop s/w for another EISC processor (SE2608, SE3208, AE6400), you must change compiler's target name.

2.1.1 Make object file

Compiler and assembler make object file form C-source code. After finish compiling and assembling, linker collects all object files to generate final execute file. First of all, a command for making object file is

$ae32000-elf-gcc –c –o <output_filename> <input_source_file>

The '-c' option announces to compiler that make object file not linking. It means the compiler compiles input source and assembler assembles an output of compiler. The <output_filename> subsequent '-o' option, informs to compiler an output name. If s/w developer does not use the '-o' option, an output object file's name of the compiler is same as source file's one.

2.1.2 Compile Path Handling

Compiler can compile with specific header files and libraries that is implemented by s/w developer. In order to include the files and libraries, developer designates a directory path. A default location of include files is '/usr/local/ae32000-elf/include' and location of libraries is '/usr/local/ae32000-elf/lib'. Compiler automatically searches these directories when compiles.

Developer can use absolute path and relative path on compile option. The absolute path indicates a path from root directory '/', and relative path is a path from current directory. Furthermore a directory path './' means current directory and '../' indicate one level above directory of current.

Next examples show a command options that handing the compile path.

Case 1. Header files and libraries locate at './project' directory.

$ae32000-elf-gcc –I./project/header/ -L./project/library/ main.c

Case 2. Header files and libraries locate at an absolute directory '/cygwin/ae32000/project'.

$ae32000-elf-gcc –I/cygwin/ae32000/project/header -L/cygwin/ae32000/project/library main.c

When user implemented header files and libraries are located in Cygwin-installed folder, developer can use command options like as case 2.

Case 3. Head files and libraries locate at an absolute directory but not Cygwin's installed folder.

$ae32000-elf-gcc –I/cygdriv/d/ae32000/project/header –L/cygdriv/d/ae32000/project/library main.c

If header files and libraries are located in other hard drive, developer can use 'cygdriv' directory. Because the cygwin can access not only installed drive, but can access other drives, the user-implemented file can be referenced by cygwin.

'-I<include directory>' option indicates the directory has header files to compile and '-L<library directory>' option indicates the library directory contains libraries. Both two options can be used multiple times as necessary.

2.1.3 Linking

Linking is a final phase of the compilation. At linking time, all of compiled object are collected by linker. Also, the liker links libraries by searching library path. Linker re-arranges all sections as described in linker script and finally makes an output file.

$ae32000-elf-gcc –o main.elf main.c crt0.o –g –O2 –Wall –lm –g –Xlinker –Tae32000.vct

Above compile command has several options. First of all, a output file is 'main.elf'. AE32000 compiler compiles the 'main.c' source file with debugging option. '-g' option lets the compiler insert debugging information at compile time. In order to use GDB, s/w developer has to use the '-g' option. Second, AE32000 assembler assembles the output of compile 'main.c' automatically. Finally, linker links two files 'main.o' and crt0.o' and math library (-lm option) with linker script ae32000.vct. A final output is main.elf. To more information of linker script, you may refer to chapter 3.

In addition, '-Wall' option make the compiler prints all of warning message. It helps to developers using '-g' and '-Wall' options for debugging. If developer do not use '–o' option, compiler's output is a.out. According to above example command, an output executable file is main.elf. Also, unless developer want to use startup file and libraries, you can use '-nostdlib', '-nostartfiles' and '-nodefaultlib' options

'-O2' option says to compiler that processes optimization the input source file. The EISC

compiler can optimize like as the GCC optimize option. Generally, many developers uses optimization level 2 when compile. GCC manual includes the optimization options and what kind of optimization works. Especially EISC compiler has several machine specific optimizations in order to high performance.

2.2 Binutils

EISC binutils is based on binutils-2.14. The EISC binutils consists with bunch of object handling application and also includes linker and assembler. GNU Binary utilities support ELF Object Code Format. The ELF format file is generally used and you want more information about that, you can refer to http://www.gnu.org/software/binutils/ .

A table below shows applications of binutils and description.

Application	Description
ae32000-elf-addr2line	Change program address to file name and line number.
ae32000-elf-ar	Make an archive and modify.
ae32000-elf-as	AE32000 assembler
ae32000-elf-ld	AE32000 linker
ae32000-elf-nm	Print symbols in object file or elf file.
ae32000-elf-objcopy	Change to order file format or copy.
ae32000-elf-objdump	Print whole information of object file or elf file.
ae32000-elf-ranlib	Make archive index and store to the archive
ae32000-elf-readelf	Print information of ELF format object file
ae32000-elf-size	Print size of each sections
ae32000-elf-strings	If a input file has string more than 4 character, prints the string
ae32000-elf-strip	Get rid of all symbols from object file or elf file.

Chapter 3
Start up

This chapter explains basics of programming on EISC system. The program details include memory map, start up code and making example code.
The start up code makes an application works on the EISC system. The job is like to start a car before driving.

This captor includes below.

3.1. Program Memory Map

3.1.1 Text section

A “.text” section includes actual instructions that can be executed on CPU. Compiler compiles a C-source code and emits the instructions. Also, the section includes read only data and constant variables. Because of the read only data is not modified or executed, the data can be placed on ROM area. Program does not access the read only data in order to execute. If a processor tries to execute that code, a program executes wrong way.

3.1.2 Data section

A “.data” section includes initialized global variables and static variables. Linker allocates the data to the data section, and the sections' VMA is explicitly described on liker script. When program execute on EISC system, start up code copies the data to actual address of the memory system.

3.1.3 Bss section

A “.bss” section includes un-initialized global variables. The un-initialized global variables are allocated on the data section like as initialized variables. The un-initialized variable does not need initilize with zero, becase of a programmer can assume the variable has zero value, start up code assigns zero to the variables.

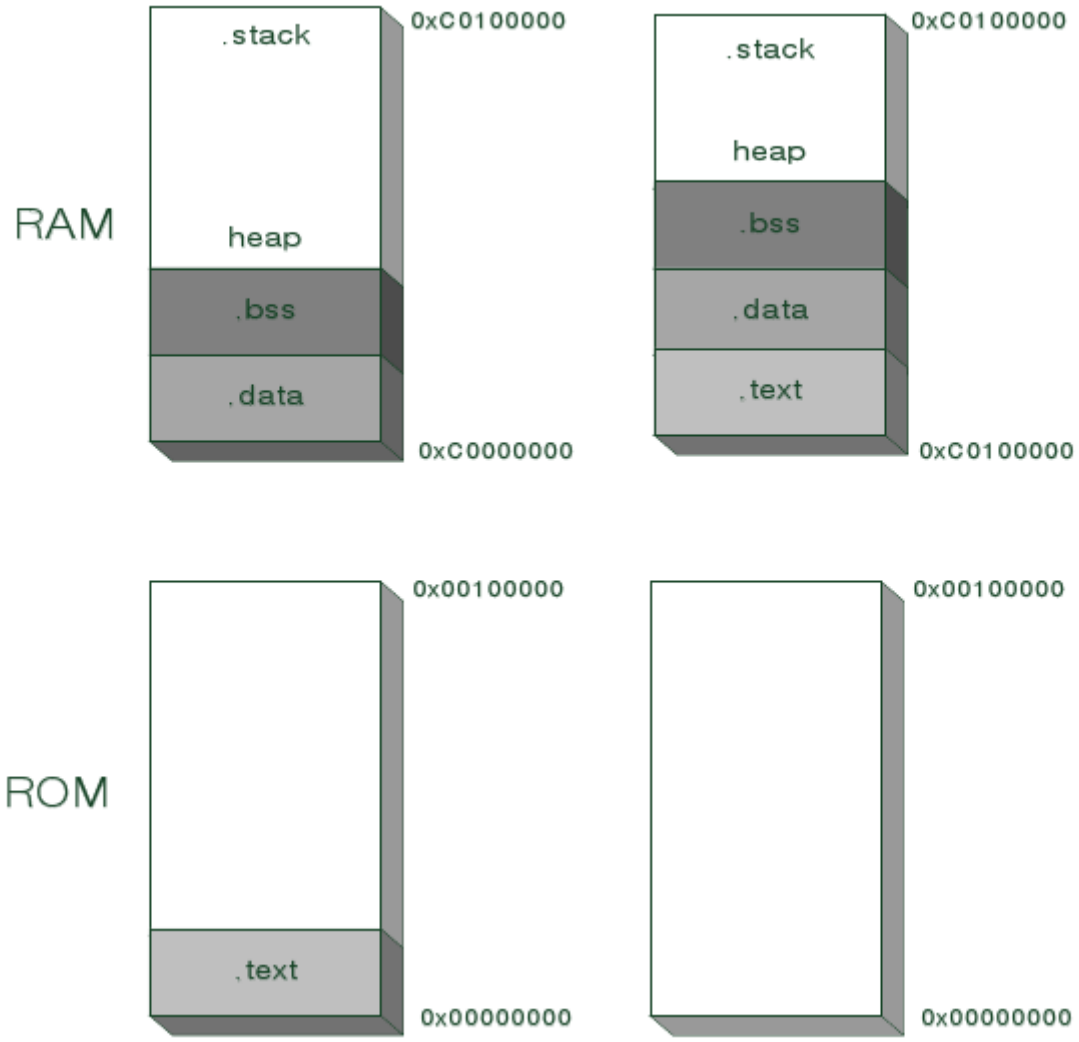
3.1.4Stack area

The EISC system uses 'downward' stack type. Due to the stack growth type, the stack's start point is an end of memory address. Stack area is temporary memory, which includes local variables, arguments, pushed register data. A stack size is determined by compiler when compile time with function by function so compiler does not know whether heap and stack collision occurs or not. The stack area is dynamically changed at run time by function call or exception handling. Program will not work if the collision is occurred at run time.

3.1.5 Heap area

A variable, which is allocated dynamically at run time, is placed in heap area. For example, if a variable is allocated by standard C functions such as malloc, calloc, etc, the variable may be allocated on heap area of a memory. As I explained 3.1.4, if the heap area's size grows up enough to collide with stack point, a program does not work. In addition to, the heap area has upward growth type.

3.1.6 Memory map



[Figure 3-1] Memory map when program is executed

For a left side of the [Figure 3-1], RAM and ROM blocks is memory map when a program is executed on ROM, and right is memory map when a program is executed on RAM. Although the ".text" section is read only code or data, the section have to allocate to RAM if the EISC system executes with RAM.

3.2 Linker Script

3.2.1 Description

Linker script includes many of information to make executable file such as output format, output architecture, start entry, included library, search directory of header file and library, memory map, section name, and so on. Liker re-arranges sections of input object files by using these information. Each sections are allocated on the memory map which defined on the liker script. Also, the linker script can provide a new symbol by 'PROVIDE' directive and the new symbol is stored to symbol table on the executable file with VMA. Linker links libraries by searching a SEARCH_DIR directory and also, they are re-arranged too.

3.2.2 Example of Linker Script

If an application executes in ROM, developer can use a linker script like as [Table 3-1]. As the table shows, the 'text section' is allocated on ROM and the 'data/bss section' is located on RAM.

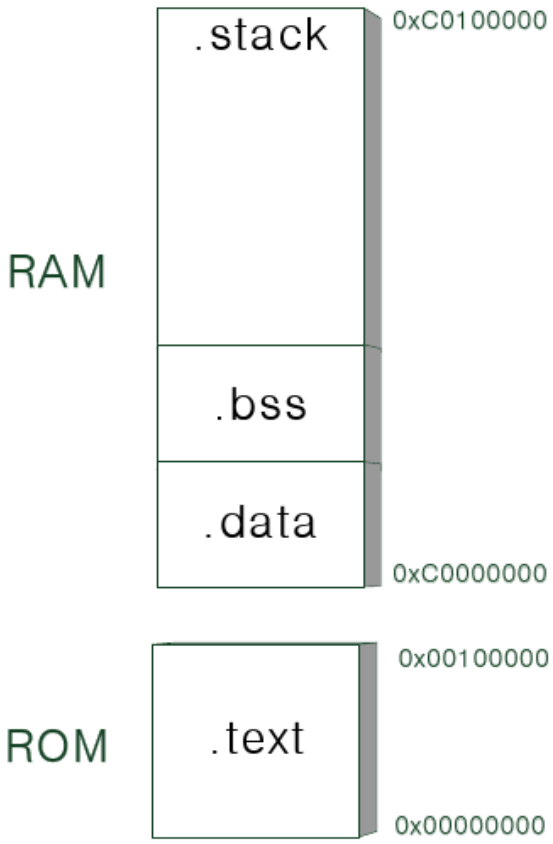
```
OUTPUT_FORMAT("elf32-ae32000-little", "elf32-ae32000-little",
                        "elf32-ae32000-little")

OUTPUT_ARCH(ae32000)

ENTRY(_START)

GROUP(-lc -lgcc -lgloss -lm)

SEARCH_DIR(.);
SEARCH_DIR(~/local/ae32000-elf/lib);

/* Do we need any of these for elf?
   __DYNAMIC = 0;    */

MEMORY
{
        rom : ORIGIN = 0x00000000          , LENGTH = 1M
        ram : ORIGIN = 0xc0000000          , LENGTH = 1M
}

SECTIONS
{
  /* Read-only sections, merged into text segment: */
  . = 0x0;
  .text      :
  {
```

```
  *(.vects)
  *(.text .text.*)
  *(.stub)
  *(.rodata .rodata.*)
  *(.rodata1)
  ___ctors = . ;
  *(.ctors)
        ___ctors_end = . ;
  ___dtors = . ;
  *(.dtors)
  ___dtors_end = . ;
  _etext = . ;

. = ALIGN(4) ;
  ___shadow_data = . ;
} > rom

PROVIDE (etext = .);
PROVIDE (___shadow_data = . );
.data :
  AT ( ADDR (.text) + SIZEOF(.text) )
{
  . = ALIGN(4);
  _data_reload = . ;
  PROVIDE(___data_reload = . );
  *(.data .data.*)
  *(.sdata .sdata.*)
  CONSTRUCTORS
} > ram
_edata  =  .;
PROVIDE (edata = .);

.bss (NOLOAD) :
                AT ( ADDR (.data) + SIZEOF(.data) + (SIZEOF(.data)&2))
{
  __bss_start = .;
PROVIDE (__bss_start = . );
        *(.dynbss)
 *(.bss .bss.*)
 *(COMMON)
} > ram

_end = . ;
PROVIDE (end = .);

._stack 0xc00ffff0 : { _stack = .; *(._stack) }
```

```
}
PROVIDE(___data_size = SIZEOF(.data));
PROVIDE(___bss_size = SIZEOF(.bss));
```

[Table 3-1] Example of Linker Script

According the linker script [Table 3-1], memory map is shown next.



[Figure 3-2] Memory map of [Table 3-1]

At the first line of the linker script, 'OUTPUT_FORMAT' indicates what kind of BFD file format used for linker. Also, there are three arguments in the directive. First argument is a default format. Second and third argument indicate little endian and big endian. AE32000 architecture uses little endian default but developers can generate a executable file as a big endian with '-EB' option.

Next line is 'OUTPUT_ARCH(ae32000)' is a definition of target architecture. You can check a target architecture of a file through '<targetname>-elf-objdump -f <input_file>' command.

'SEARCH_DIR(/usr/local/ae32000-elf/lib)' is a definition of searching directory to link libraries for linker. It is same work with using command option '-L<library directory path>' when compile.

'MEMORY' indicates a memory map of the target architecture. According to [Table 3-1], there are two memory types ROM and RAM and we can know about the start address and size.

'SECTIONS' defines a memory layout of all section to generate an output file. Loader copies the sections to actual memory from the output file. The linker script can have only the 'SECTIONS' directive. According to the linker script shown in [Table 3-1], the 'text section' consists of '.vects', '.stub', '.rodata', '.ctors' and '.dtros' sections.

In addition '.=0x0' means the '.text' section starts with memory address 0x0. At the last of definition '.text', you can see '>rom'. That means the '.text' section allocates to ROM of memory. In order hands, '.data', '.bss' sections are allocate to RAM.

__ctors, __ctors_end, __dtors and __dtors_end are constructor and destructor of C++ application. The constructor is called when executes __main function and the destructor is called when executes exit or atexit function called.

Finally, linker scripts can add symbols by using 'PROVIDE' directive. The symbols added by linker script cannot be use in C/C++ source file. If developer uses the symbol, the EISC compiler occurs re-definition error. 'PROVIDE(__data_size = SIZEOF(.data))' means the output file contains a symbol named '__data_size' and its value is a size of '.data' section.

3.3 crt0.S

3.3.1 Introduction

CRT0 means C Runtime Library 0 and when a system executes an application, the CRT0 is executed first. The crt0 includes initialize all of system environment such as stack pointer register, general register as well as device registers.

crt0.S of EISC system works bunch of jobs explained blow.

- Initialize stack pointer register
- Copy the data section
- Initialize bss section with zero
- Initialize constructor when use C++ code.
- Calls main().
- Calls exit().

3.3.2 Source code and Description

After RESET the EISC system, program enters into the crt0.S. First of all, the crt0 initializes stack pointer register %SP. A value of stack pointer register is defined in linker script.

```
linker scripter AE32000.vct
            … …
            .debug_typenames 0 :{*(.debug_typenames)}
            .debug_varnames 0 : {*(.debug_varnames)}
            /*These must appear regardless of . */
           ._stack 0xc00ffff0 : { _stack = .; *(._stack) }
}

PROVIDE(___data_size = SIZEOF(.data));
PROVIDE(___bss_size = SIZEOF(.bss));
```

[Table 3-2] A part of linker script

From [Table 3-2], the stack address is initialized with a value 0xc00ffff0. The address of stack is a last of memory due to EISC system has a downward type.

```
_START:
/* initialize stack pointer */
            ldi _stack-8, %r8
            mov %r8,%sp
……
```

[Table 3-3] A part of crt0.S to initialize %SP

The %R8 register's value will be _stack -8, and the %SP is set up with that value.

```
jal ___main    # A function call which initializes bss and data section
jal _main      # Calls a main()
```

[Table 3.4] A part of crt0.S to initialize memory and calls main().

3.4 crt1.c

3.4.1 Introduction

crt1.c C source file is collection of the initialization functions which called by crt0.S.

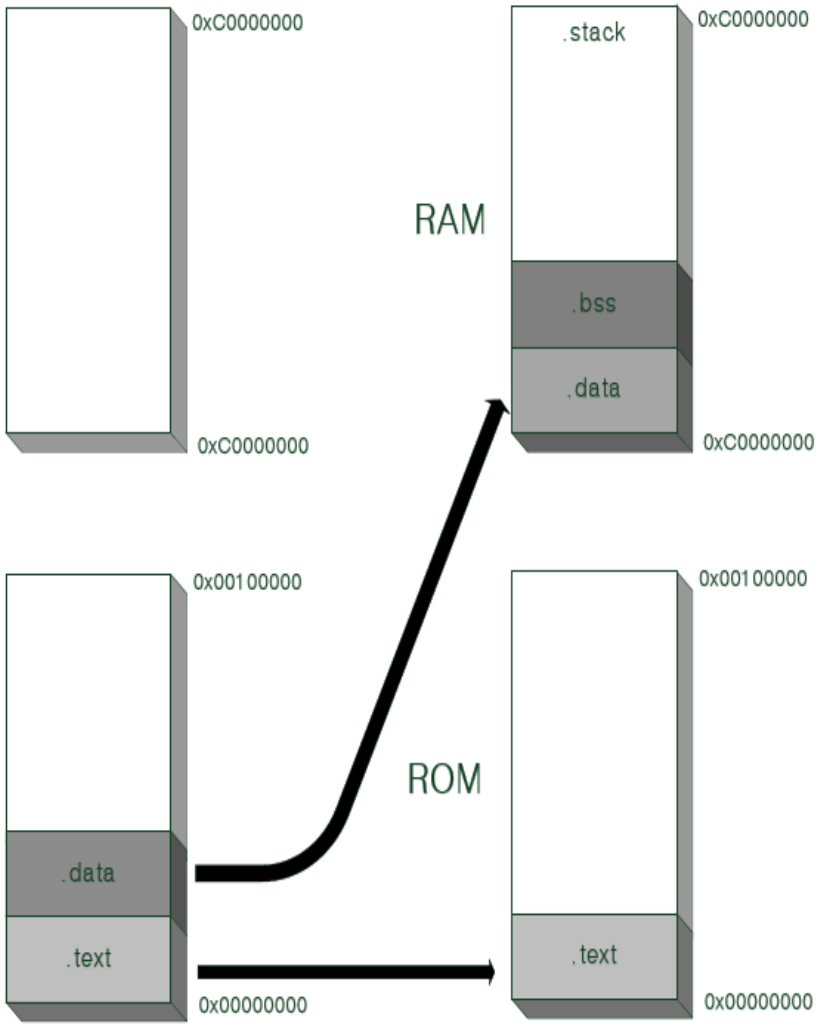
3.4.2 Source code and Description

```
extern unsigned char __shadow_data[];
extern unsigned char __data_reload[];
extern const     int   __data_size[];
extern const     char _bss_start;
extern const     int __bss_size[];

void START();
void (*vector_table[])(void) __attribute__((section (".vects")))= {
    START               , // RESET VECTOR
};

void __main ()
{
  volatile unsigned char * srcptr ;
  volatile unsigned char * destptr ;
  int count ;
  typedef void (*pfunc) ();
  extern pfunc __ctors[];
  extern pfunc __ctors_end[];
  pfunc *p;

  /* data section initialization with default value */
  srcptr = (char *)&__shadow_data;                  // Start address of data section on ROM
  destptr = (char *)&__data_reload;                 // Start address of data section on RAM
  count    = (int)&__data_size;                        // Size of data section
  memcpy(destptr, srcptr, count);               // Copy a data of data section


  /* bss section initialization with zero value */
  srcptr = (char *)&_bss_start;                          // Start address of bss section on RAM
  count = (int)&__bss_size;                          // Size of bss section
  while(count--)*srcptr++ = 0;                          // Initializing the bss section


  for (p = __ctors_end; p > __ctors; )        // Initializing constructors
```

```
    (*--p) ();
}
```

[Table 3-5] Full source of crt1.c

According to the [Table 3-5], a function __main() is called by crt0.S and that initializes data/bss section.



[Figure 3-3] Before and after executing start up code

As you can see the [Figure 3-3], a binary file is located in ROM. After initializing data/bss section, the data are copied to RAM area, which can be read/written. Data of bss section is initialized with zero value. Between the bss section and .stack area is used as the heap.

3.5 Make an example code

3.5.1 Print “Hello World!”

In order to print out “Hello World!”, we need four files crt0.S, crt1.c, main.c and ae32000.vct.

3.5.2 ae32000.vct for the example

It is same source code with [Table 3-1]

3.5.3 crt0.S of the example

```
##-------------------------------------------------------------
        .file           "crt0.S"
##-------------------------------------------------------------
        .section .text
        .global         _START
_START:
        ldi _stack-8,%r8
        mov %r8,%sp

.L0:
        jal ___main
        jal _main


.section ._stack
_stack:  .long          1
```

[Table 3-6] crt0.S source code

3.5.4 main program exmaple code

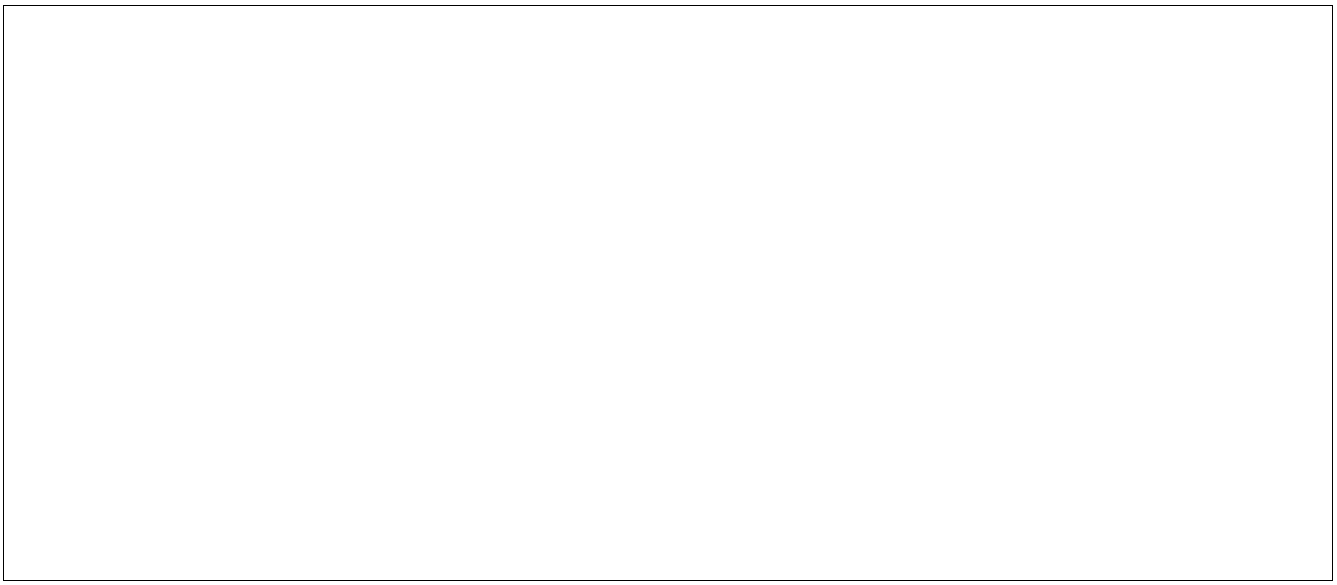
```
#include <stdio.h>
#include <stdarg.h>
#include <string.h>
#include <stdlib.h>
extern void START();
const fp HardwareVector[] __attribute__((section (".vects")))= {
                START, /* V00 : Reset Vector */
                0
};
int main()
```

```
{
        printf (“Hello World!\r\n”);
        while(1);
}
```

[Table 3-7] main.c source code

3.5.5 Execution result on GDB simulator.

Command : ae32000-elf-gcc -o main.elf crt0.S crt1.c -Xlinker -Tae32000.vct



[Figure 3-4] Screen of example execution result

Chapter 4
C/C++ and Assembly language programming

This chapter introduces a method of C/C++ and assembly language programming. In addition to, for the EISC system, specific information is described such as special register handling, ABI and built-in-functions.

This captor includes below.

4.1 Using Inline Assembly

The EISC compiler cannot know about the co-processor information. To access the co-processor, developer can use a inline assembly method. Basically, the compiler cannot access to the processor directly due to this reason.

4.1.1 Features of the inline assembly

C and C ++ and inline assembly is extremely flexible by supporting inter-working.
Operand registers is a formula of an arbitrary expression of C or C ++. The inline assembly is also a complex command expansion and optimizes the assembly language code.

```
int main(void)
{
   asm("1di 0x10000,%r1");
   asm("ld(%r1,0x200),%r2");
   asm("add %r1, %r2, %r3");
}
```

[Table 4-1] Examples of using an inline assembly command (AE32000)

When developer uses optimization option during compile, the inline assembly language can be affect from the optimization routine.

4.1.2 Features of an embedded assembler

If you use an assembly language when implements a specific application, you can access target processor, memory, variables with no limitation. The assembly code is compiled separately with C code. The inlined assembly code is inserted into the output of compiler.

4.2 Examples for Inline assembly: using C variable in assembly code

4.2.1 Use method of C language variable

When developer implements assembly codes in C source, standard C language supports accessing variables from the assembly code. Syntax of the inline assembly code is shown below.

```
asm( assembler template
  : output operand /* optional */
  : input operand /* optional */
  : list of clobbered registers /* optional */
);
```

[Table 4-2] Syntax of inline assembly

According to the inline assembly syntax, the inline statement starts with 'asm' directive. The assembler template consists of opcode and operand string. The operand string represents '%<number>'. The number of operand string indicates defined variable's number. That means, to use first define variable, the operand string may be '%0' and second is '%1'.

':' separates the input/output operand groups. The input/output operands are composed of a kind and a name of variable. Compiler variable class defines a kind of variables.
'g' indicates global variable and 'r' is a register, 'i' is an immediate constant and 'm' is a memory variable. If a variable to use as output, '=' is attached to the variable kind.

'Clobbered register' indicates a hardware register that used in the assembly template. For example, in a statement 'asm("mov %0, %**%R0**" : : "r"(a) : "**R0**")', register R0 is touched by compiler.

4.2.2 Example of inline assembly code

```
#include <stdio.h>
__main_init(){};
int main(void)
{
        int c, d;

        c = 0x02;
        d = 0x30;
        asm("TEST:");
        asm("ld %0, %%r1" :: "g"(d));
        asm("add %r0, %%r1" :: "r"(c));
        asm("st %%r1, %r0" : "=g"(d):);
```

```
        printf("\n %x", d);
}
```

[Table 4-3] A example of inline assembly code

4.3 A Function call between C/C++ and Assembly language

This section explains calling methods the C and assembly language and how to call in C++.

4.3.1 Approaching to assembly function in C source code

This paragraph, we can found a method to call and how to use functions that consisting of assembly commands inside C source code.

To call assembly functions that declared and defined at another file in C program source code, like [table 4-4], you should start with '_' character for symbol and declare as a Global in assembly source code.

- Assembly source

```
# A function to add 1 to 10
            .global _add    #Function definition
_add: #Label
            push %r0, %r1
            ldi 0, %r0
            ldi 10, %r1
            ldi 0, %r8
add_loop: # Branch target
            add 1, %r0 # Increase by 1
            add %r0, %r8
            cmp %r0, %r1
            jnz add_loop # Loop until R0 is 10
            pop %r0, %r1
            jplr
```

[Table 4-4] Assembly source code of _add function

- C source

```
#include <stdio.h>
__main_init(){};
extern int add(); //External function
int main(void)
```

```
{
        int c:
        c = add();   //Calls the function add
        printf("₩n %d",c);
}
```

[Table 4-5] C source code to call a _add function

Examining above [table 4-4], you can found using called with function in [table 4-5] c source code after defined with symbol '_add' and declared as global just next line. At this time, there is relevance between two files and they must be linked together of course.

To receive a variable from turned into the assembly function to c language function or to return a value to c language function, you must follow a factor transmission method among functions in c language.

```
c0700000 <_add>:


.global _add   #Function definition
_add: # Label
push %r0, %r1
c0700000: 03 b0 push %R0, %R1


c0700002 <L0┌L1>:
ldi 0, %r0
c0700002: 00 a0 ldi 0x0 %R0


c0700004 <L0┌L2>:
ldi 10, %r1
c0700004: 0a a1 ldi 0xA %R1


c0700006 <L0┌L3>:
ldi 0, %r8
c0700006: 00 a8 ldi 0x0 %R8
```

```
c0700008 <L0┌L4>:
add_loop: # Branch target of loop end
add 1, %r0 #Increases by 1
c0700008: 00 40 leri 0x0 (0x0)
c070000a: 01 b8 add 0x1 %R0

c070000c <L0┌L5>:
add %r0, %r8
c070000c: 80 b8 add %R0 %R8

c070000e <L0┌L6>:
cmp %r0, %r1
c070000e: 10 bf cmp %R0 %R1

c0700010 <L0┌L7>:
jnz add_loop
c0700010: ff 41 leri 0x1FF (0x1FF)
c0700012: ff 7f leri 0x3FFF (0x7FFFFF)
c0700014: f9 d4 jnz c0700008 <L0┌L4>

c0700016 <L0┌L8>:
pop %r0, %r1
c0700016: 03 b1 pop %R0, %R1

c0700018 <L0┌L9>:
jplr
c0700018: a0 e0 jplr
                                    ……..
#include <stdio.h>
__main_init(){};
c070015a: 20 b4 push %lr
c070015c: 60 b0 push %R5, %R6
c070015e: d6 e1 lea ( %SP ) %R6
```

```
c0700160 <.LM2>:
c0700160: c6 e1 lea ( %R6 ) %SP
c0700162: 60 b1 pop %R5, %R6
c0700164: 40 b5 pop %pc
c0700166 <_main>:
extern int add(); External function
int main(void)
{
c0700166: 20 b4 push %lr
c0700168: 60 b0 push %R5, %R6
c070016a: d6 e1 lea ( %SP ) %R6
c070016c: fc b6 lea ( %SP 0xFFFFFFF0 ) %SP

c070016e <.LM4>:
int c;
c = add(); // Function call
c070016e: ff 41 leri 0x1FF (0x1FF)
c0700170: ff 7f leri 0x3FFF (0x7FFFFF)
c0700172: 46 df jal c0700000 <_add>
c0700174: ff 7f leri 0x3FFF (0xFFFFFFFF)
c0700176: 36 18 st %R8 , ( %R6 +  0xFFFFFFFC )

c0700178 <.LM5>:
printf("₩n %d",c);
c0700178: ff 7f leri 0x3FFF (0xFFFFFFFF)
c070017a: 36 09 ld ( %R6 +  0xFFFFFFFC ) %R9
c070017c: 1c 70 leri 0x301C (0xFFFFF01C)
c070017e: bd 48 leri 0x8BD (0xFC0708BD)
c0700180: 02 a8 ldi 0xC0708BD2 %R8
c0700182: 00 40 leri 0x0 (0x0)
c0700184: 00 40 leri 0x0 (0x0)
c0700186: 21 df jal c07001ca <_printf>
```

```
c0700188 <.LM6>:
}
c0700188: c6 e1 lea ( %R6 ) %SP
c070018a: 60 b1 pop %R5, %R6
c070018c: 40 b5 pop %pc
```

[Table 4-6] result disassemble of [table 4-4,4-5] source code (AE32000)

At [table 4-6], you can verify the right that call assembly function of the files in C program source code.

4.3.2 General rules for call among languages

By using C call rule. In c++, you can designate to include c linkage with declaring non-member function into extern "c". Including c linkage means a symbol name which define a function isn't change. using c linkage, you can call at ather languages after materialize one language. Declared function with Extern "c" can't overload.

4.3.3 examples call among languages

- Call a C function in C++

[table 4-7] and [table 4-8] are displaying calling a method of c in c++.

```
extern "C" void cfunc(int i*);
// declare the C function to be called from C++
int f(){
int i=0;
        cfunc(&i);
        return i * 3;
}
```

[Table 4-7] C function calls in C++

```
void cfunc(int *p) {
        *p += 5;
}
```

[Table 4-8] A function defined in C

- call a C++ function in C

[table 4-9] and [table 4-10] show calling method of c++ in c.

```
extern "C" void cppfunc(int *p);
void cppfunc(int *p)
{
        *p += 5;
}
```

[Table 4-9] A function define for call in C++

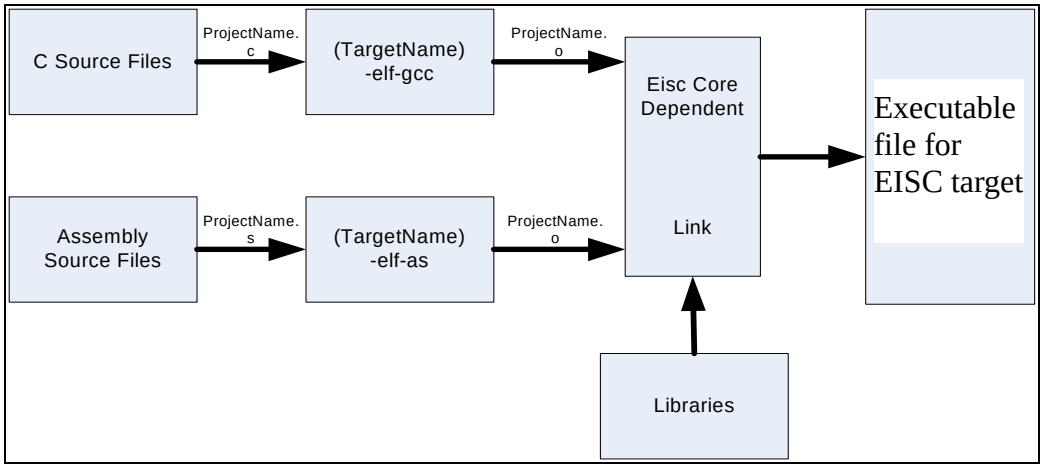
```
extern void cppfunc(int *i);
/* Declaration of the C++ function to be called from C */
int f(void) {
int i=0;
        cppfunc(&i); /* call 'cppfunc' so it */
        return i * 3;
}
```

[Table 4-10] Declare and call of function in C

4.4 Standard Function Call

4.4.1 Standard function call

In order to get a high performance by reducing code size, developers sometimes implements an application with mixing C/C++ and assembly language. [Figure 4-1] shows flows of compiling and



[Figure 4-1] Application creation for EISC target

This section handles argument passing, calling method and registers usage.

4.4.2 Registers

EISC processors have 8 to 16 general purpose registers and 4 to 7 special purpose registers. The registers' size is 32bit or 64bit. ISA of EISC processor has 16-bit size of instruction.

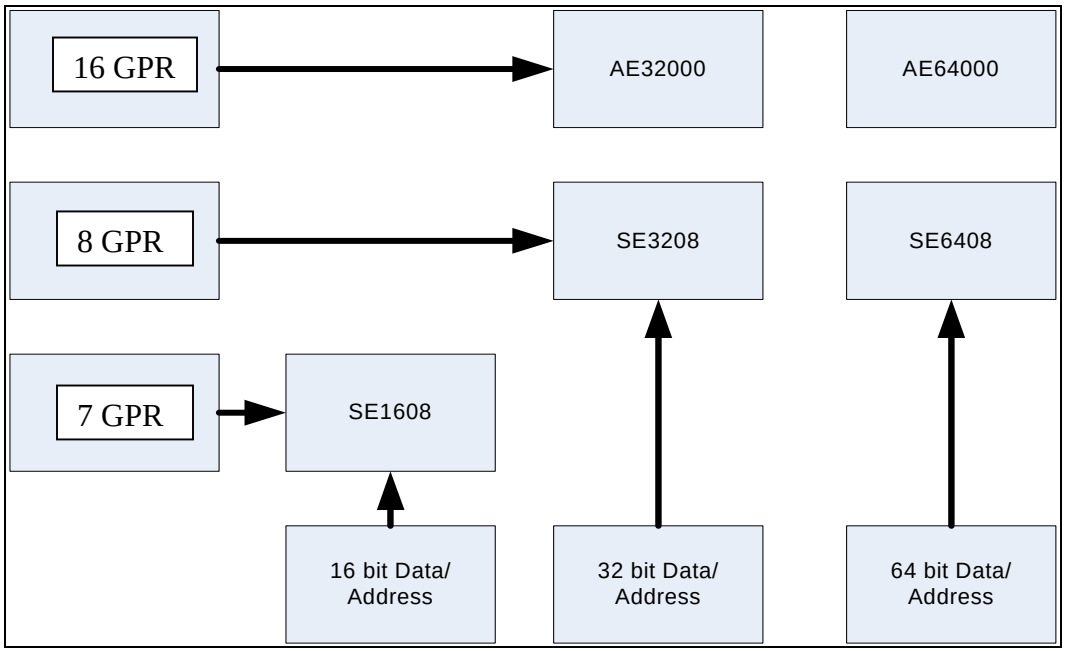
Register	EISC Core	Description
R0 - R7	SE1608	%r0 ~ %r6 : GPR, %r7 : Stack Pointer(SP)
	SE3208	General Purpose Register
R8 - R15	Up to AE32000	General Purpose Register
PC	All of EISC processor	Program Counter
SP	All of EISC processor	Stack Pointer
USP	Up to SE3208	User Stack Pointer
SSP	Up to SE3208	Supervisor Stack Pointer
ISP	Up to SE3208	OSI Stack Pointer
LR	Up to AE32000	Link Register

ER	All of EISC processor	Extension Register
SR	All of EISC processor	Status Register

[Table 4-11] EISC processors' registers

Bit	Core	Description
63 – 32	Up to AE64000	Reserved, always 0
31 - 16	Up to AE32000	Reserved, always 0
15	Down to SE3208	Reserved, always 0
	Up to AE32000	Bit[15,9] : Processor Mode 00 : Supervisor Mode 01 : OSI Mode 10 : User Mode
14	All of EISC processor	NMI Enable
13	All of EISC processor	Interrupt Enable
12	All of EISC processor	0 : Auto-vectored Interrupt Enable
11	SE1608	Reserved, always 0
	Up to SE3208	Extension Flag
10	Down to SE3208	Reserved, always 0
	Up to AE32000	Endianness, Read only. Set/clear on power-on Configuration 0 : Little Endian 1 : Big Endian
9	Down to SE3208	Reserved, always 0
	Up to AE32000	Processor mode. See bit 15
8	Down to SE3208	Reserved, always 0
	Up to AE32000	Lock Flag
7	All of EISC processor	Carry Flag
6	All of EISC processor	Zero Flag
5	All of EISC processor	Sign Flag
4	All of EISC processor	Overflow Flag
3-0	All of EISC processor	Reserved, always 0

[Table 4-12] EISC registers usage



[Figure 4-2] GPR and address bit size of EISC processor

A status register of EISC processor depends its processor type. If you want more information about the SR register, you should refer to a h/w reference manual.

EISC SE1608, AE3208 use %R0, %R1 registers and AE32000, AE64000 use %R8, %R9 registers for arguments.
Register %R6 is used as frame pointer register and %R7 is temporary register.

4.4.3 Function call

When a caller calls callee, the caller passes arguments to the callee. The argument passing is done through the argument registers. As I explained, in the AE32000 architecture uses %R8 and %R9. If the caller has to pass arguments more than three, after third argument is passed through stack space.
Callee uses %R8 register as a return value at the end of function.

① An example codes that passes 2 arguments and returns one value.

```
#include <stdio.h>
__main_init(){};
int Sub_Function(char one, int two)
{
        int result;
        result = one + two;
        return result;
```

```
}
int main(void)
{
        char one;
        int two, return_value;
        one = 1;
        two = 2;
        return_value = Sub_Function(one, two);
        return return_value;
}
```

[Table 4-13] Passes 2 arg and returns one value

[Table 4-13] The function ‘Sub_Function’ passes two arguments ‘one’ and ‘two’ and returns summation of the two. We can actual instructions by disassembling.

```
$ ae32000-elf-objdump -disassemble-all -S filename.elf > filename.dis
```

```
output/test_p.elf:      file format elf32-ae32000-little
output/test_p.elf
architecture: ae32000, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0xc0700140

Program Header:
LOAD off    0x00001000 vaddr    0xc0700000 paddr 0xc0700000 align
2**12
        filesz    0x000001c0 memsz 0x000001c0 flags rwx

Sections:
Idx Name              Size        VMA         LMA         File off    Algn
  0 .text             000001c0    c0700000    c0700000    00001000    2**1
CONTENTS, ALLOC, LOAD, READONLY, CODE
…….
_START:
/* initialize stack pointer : move unline from GPR upline*/
        ldi _stack-8,%r8
c0700140:    1f 70          leri    0x301F        (0xFFFFF01F)
                    ……

c0700160 <.LM2>:
c0700160:    c6 e1          lea     ( %R6    ) %SP
c0700162:    60 b1          pop     %R5, %R6
c0700164:    40 b5          pop     %pc

c0700166 <_Sub_Function>:
int Sub_Function(char one, int two)
```

```
{
c0700166:      20 b4            push    %lr

c0700174 <.LM4>:
c070018a <_main>:
int main(void)
{
c0700174:      02 40            leri    0x2     (0x2)
c0700176:      06 29            ldb     ( %R6   + 0x10 ) %R9
c0700178:      56 08            ld      ( %R6   + 0x14 ) %R8
c070017a:      89 b8            add     %R9    %R8
c070017c:      ff 7f            leri    0x3FFF          (0xFFFFFFFF)
c070017e:      36 18            st      %R8    , ( %R6          + 0xFFFFFFFC )

c0700180 <.LM5>:
        return result;
c0700180:      ff 7f            leri    0x3FFF          (0xFFFFFFFF)
c0700182:      36 08            ld      ( %R6   + 0xFFFFFFFC ) %R8

c0700184 <.LM6>:
}
```

```
c0700198:    09 38          stb     %R8   , ( %R9          + 0x0 )

c070019a <.LM9>:
        two = 2;
c070019a:    02 a8          ldi     0x2     %R8
c070019c:    ff 7f          leri    0x3FFF          (0xFFFFFFFF)
c070019e:    26 18          st      %R8   , ( %R6          + 0xFFFFFFF8 )

c07001a0 <.LM10>:
        return_value = Sub_Function(one, two);
c07001a0:    86 e4          lea     ( %R6     ) %R8
c07001a2:    df c8          addq    0xffffffff,       %R8
c07001a4:    08 28          ldb     ( %R8   + 0x0 ) %R8
c07001a6:    ff 7f          leri    0x3FFF          (0xFFFFFFFF)
c07001a8:    26 09          ld      ( %R6   + 0xFFFFFFF8 ) %R9
c07001aa:    ff 41          leri    0x1FF (0x1FF)
c07001ac:    ff 7f          leri    0x3FFF          (0x7FFFFF)
c07001ae:    db df          jal     c0700166 <_Sub_Function>
c07001b0:    ff 7f          leri    0x3FFF          (0xFFFFFFFF)
c07001b2:    16 18          st      %R8   , ( %R6          + 0xFFFFFFF4 )

c07001b4 <.LM11>:
        return return_value;
c07001b4:    ff 7f          leri    0x3FFF          (0xFFFFFFFF)
c07001b6:    16 08          ld      ( %R6   + 0xFFFFFFF4 ) %R8

c07001b8 <.LM12>:
}
c07001b8:    c6 e1          lea     ( %R6     ) %SP
c07001ba:    60 b1          pop     %R5, %R6
c07001bc:    40 b5          pop     %pc

c07001be <___ctors>:
        ...
Disassembly of section .data:
```

[Table 4-14] Disassemble result of [Table 4-13]

As the [Table 4-14], function ‘main’ sends two registers %R8, %R9 as arguments to ‘Sub_Function’. The ‘Sub_Function’ returns a result value through the register %R8.

② An example codes that passes 3 arguments and returns structure.

```
#include <stdio.h>
__main_init(){};
typedef struct {
```

```
        int sum, car;
        char str[20];
} TOTAL;
TOTAL Sub_Function(char one, int two, int three)
{
        TOTAL a;
        a.sum = two + one;
        a.car = a.sum + 100 + three;
        a.str[0] = 'a';
        a.str[1] = 'b';
        a.str[2] = 'c';
        a.str[3] = 'd';
        a.str[4] = 'e';
        a.str[5] = 'f';
        a.str[6] = 0;
        return a;
}
int main(void)
{
        char one;
        int two, three;
        TOTAL b;
        one = 1;
        two = 2;
        three = 3;
        b = Sub_Function(one, two, three);
        return b.sum;
}
```

[Table 4-15] Passes 3 arguments and returns one value

Disassembling result is shown below.

```
output/test_p.elf:      file format elf32-ae32000-little
output/test_p.elf
architecture: ae32000, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0xc0700140

Program Header:
LOAD off    0x00001000 vaddr 0xc0700000 paddr 0xc0700000 align 2**12
     filesz 0x000002f4 memsz 0x000002f4 flags rwx

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         000002f4  c0700000  c0700000  00001000  2**1
```

```
CONTENTS, ALLOC, LOAD, READONLY, CODE
……
_START:
/* initialize stack pointer : move unline from GPR upline*/
        ldi _stack-8,%r8
c0700140:    1f 70          leri    0x301F          (0xFFFFF01F)
c0700142:    fe 7f          leri    0x3FFE          (0xFC07FFFE)
c0700144:    08 a8          ldi     0xC07FFFE8  %R8

c0700146 <L0┌L1>:
        mov %r8,%sp
c0700146:    c8 e1          lea     ( %R8     ) %SP

c0700148 <L0┌L2>:
.L0:
        jal ___main2
c0700148:    ff 41          leri    0x1FF  (0x1FF)
c070014a:    ff 7f          leri    0x3FFF          (0x7FFFFF)
c070014c:    bd df          jal     c07000c8 <___main2>

c070014e <L0┌L3>:
        jal ___main_init
c070014e:    00 40          leri    0x0     (0x0)
c0700150:    00 40          leri    0x0     (0x0)
c0700152:    03 df          jal     c070015a <___main_init>

c0700154 <L0┌L4>:

        jal _main
c0700154:    00 40          leri    0x0     (0x0)
c0700156:    00 40          leri    0x0     (0x0)
c0700158:    61 df          jal     c070021c <_main>

c070015a <___main_init>:
#include <stdio.h>
__main_init(){};
c070015a:    20 b4          push    %lr
c070015c:    60 b0          push    %R5, %R6
c070015e:    d6 e1          lea     ( %SP    ) %R6

c0700160 <.LM2>:
c0700160:    c6 e1          lea     ( %R6     ) %SP
c0700162:    60 b1          pop     %R5, %R6
c0700164:    40 b5          pop     %pc

c0700166 <_Sub_Function>:
```

```
typedef struct {
int sum, car;
char str[20];
} TOTAL;
TOTAL Sub_Function(char one, int two, int three)
{
c0700166:   20 b4        push   %lr
c0700168:   63 b0        push   %R0, %R1, %R5, %R6
c070016a:   d6 e1        lea    ( %SP    ) %R6
c070016c:   f5 b6        lea    ( %SP  0xFFFFFFD4    ) %SP
c070016e:   18 e4        lea    ( %R8     ) %R1
c0700170:   89 e4        lea    ( %R9     ) %R8
c0700172:   03 40        leri   0x3     (0x3)
c0700174:   46 38        stb    %R8    , ( %R6          + 0x1C )

……
c07001fc <.LM13>:
return a;
c07001fc:   86 e4        lea    ( %R6     ) %R8
c07001fe:   fe 7f        leri   0x3FFE         (0xFFFFFFFE)
c0700200:   84 b8                add    0xFFFFFFE4        %R8
c0700202:   91 e4        lea    ( %R1     ) %R9
c0700204:   08 e4        lea    ( %R8     ) %R0
c0700206:   1c a8        ldi    0x1C   %R8
c0700208:   83 98        st     %R8    , ( %SP          + 0xC )
c070020a:   89 e4        lea    ( %R9     ) %R8
c070020c:   90 e4        lea    ( %R0     ) %R9
c070020e:   00 40        leri   0x0     (0x0)
c0700210:   00 40        leri   0x0     (0x0)
c0700212:   2a df        jal    c0700268 <_memcpy>

c0700214 <.LM14>:
}
c0700214:   81 e4        lea    ( %R1     ) %R8
c0700216:   c6 e1        lea    ( %R6     ) %SP
c0700218:   63 b1        pop    %R0, %R1, %R5, %R6
c070021a:   40 b5        pop    %pc

c070021c <_main>:
int main(void)
{
c070021c:   20 b4        push   %lr
c070021e:   61 b0        push   %R0, %R5, %R6
c0700220:   d6 e1        lea    ( %SP    ) %R6
c0700222:   f1 b6        lea    ( %SP  0xFFFFFFC4    ) %SP
```

```
c0700224 <.LM16>:
        char one;
int two, three;
TOTAL b;
TOTAL b;
one = 1;
c0700224:    96 e4          lea     ( %R6    ) %R9
c0700226:    df c9          addq    0xffffffff,      %R9
c0700228:    01 a8          ldi     0x1     %R8
c070022a:    09 38          stb     %R8    , ( %R9          + 0x0 )

c070022c <.LM17>:
two = 2;
c070022c:    02 a8          ldi     0x2     %R8
c070022e:    ff 7f          leri    0x3FFF          (0xFFFFFFFF)
c0700230:    26 18          st      %R8    , ( %R6          + 0xFFFFFFF8 )

c0700232 <.LM18>:
three = 3;
c0700232:    03 a8          ldi     0x3     %R8
c0700234:    ff 7f          leri    0x3FFF          (0xFFFFFFFF)
c0700236:    16 18          st      %R8    , ( %R6          + 0xFFFFFFF4 )

c0700238 <.LM19>:
b = Sub_Function(one, two, three);
c0700238:    96 e4          lea     ( %R6    ) %R9
c070023a:    fd 7f          leri    0x3FFD          (0xFFFFFFFD)
c070023c:    98 b8          add     0xFFFFFFD8 %R9
c070023e:    86 e4          lea     ( %R6    ) %R8
c0700240:    df c8          addq    0xffffffff,      %R8
c0700242:    08 20          ldb     ( %R8    + 0x0 ) %R0
c0700244:    ff 7f          leri    0x3FFF          (0xFFFFFFFF)
c0700246:    26 08          ld      ( %R6    + 0xFFFFFFF8 ) %R8
c0700248:    83 98          st      %R8    , ( %SP          + 0xC )
c070024a:    ff 7f          leri    0x3FFF          (0xFFFFFFFF)
c070024c:    16 08          ld      ( %R6    + 0xFFFFFFF4 ) %R8
c070024e:    84 98          st      %R8    , ( %SP          + 0x10 )
c0700250:    89 e4          lea     ( %R9    ) %R8
c0700252:    90 e4          lea     ( %R0    ) %R9
c0700254:    ff 41          leri    0x1FF (0x1FF)
c0700256:    ff 7f          leri    0x3FFF          (0x7FFFFF)
c0700258:    86 df          jal     c0700166 <_Sub_Function>

c070025a <.LM20>:
```

```
return b.sum;
c070025a:    86 e4          lea    ( %R6    ) %R8
c070025c:    fd 7f          leri   0x3FFD        (0xFFFFFFFD)
c070025e:    88 b8          add    0xFFFFFFD8 %R8
c0700260:    08 08          ld     ( %R8   + 0x0 ) %R8

c0700262 <.LM21>:
}c0700262:   c6 e1          lea    ( %R6    ) %SP
c0700264:    61 b1          pop    %R0, %R5, %R6
c0700266:    40 b5          pop    %pc

……
c07002d0 <.L15>:
c07002d0:    29 e4          lea    ( %R9    ) %R2

c07002d2 <.LM16>:
c07002d2:    df c1          addq   0xffffffff,      %R1
c07002d4:    ff c1          cmpq   0xffffffff,      %R1
c07002d6:    00 40          leri   0x0     (0x0)
c07002d8:    00 40          leri   0x0     (0x0)
c07002da:    09 d5          jz     c07002ee <.L17>

c07002dc <.L21>:
c07002dc:    00 28          ldb    ( %R0   + 0x0 ) %R8
c07002de:    c1 c0          addq   0x1,    %R0
c07002e0:    02 38          stb    %R8   , ( %R2          + 0x0 )
c07002e2:    c1 c2          addq   0x1,    %R2

c07002e4 <.L2>:
c07002e4:    df c1          addq   0xffffffff,      %R1
c07002e6:    ff c1          cmpq   0xffffffff,      %R1
c07002e8:    ff 41          leri   0x1FF (0x1FF)
c07002ea:    ff 7f          leri   0x3FFF        (0x7FFFFF)
c07002ec:    f7 d4          jnz    c07002dc <.L21>

c07002ee <.L17>:
c07002ee:    83 e4          lea    ( %R3    ) %R8
c07002f0:    0f b1          pop    %R0, %R1, %R2, %R3
c07002f2:    40 b5          pop    %pc
Disassembly of section .data:
```

[Table 4-16] Disassemble result of [Table. 4-15]

The main function passes two arguments ‘one’ and ‘two’ by using %R8 and %R9 registers, and argument ‘three’ is stored to stack and the ‘Sub_Function’ loads the argument to use.

The ‘Sub_Functon’ returns a pointer value of structure.

③ An example code that passes and receives structure data type.

```
#include <stdio.h>
__main_init(){};
typedef struct {
        int sum;
        int car;
        char str[7];
} TOTAL;
TOTAL Sub_Function(char one, int two, int three, TOTAL b)
{
        TOTAL a;
        a.sum = two + one + b.sum;
        a.car = a.sum + 100 + three;
        a.str[0] = 'a';
        a.str[1] = 'b';
        a.str[2] = 'c';
        a.str[3] = 'd';
        a.str[4] = 'e';
        a.str[5] = 'f';
        a.str[6] = 0;
        return a;
}
int main(void)
{
        char one;
        int two, three;
        TOTAL b;
        one = 1;
        two = 2;
        three = 3;
        b.sum = 15;
        b = Sub_Function(one, two, three, b);
        return b.sum;
}
```

[Table 4-17] Passes and returns structure

Disassembling result is shown below.

```
output/test_p.elf:      file format elf32-ae32000-little
output/test_p.elf
architecture: ae32000, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0xc0700140
```

```
Program Header:
LOAD off      0x00001000 vaddr 0xc0700000 paddr 0xc0700000 align 2**12
         filesz 0x000002b8 memsz 0x000002b8 flags rwx

Sections:
Idx Name              Size        VMA         LMA         File off    Algn
  0 .text             000002b8    c0700000    c0700000    00001000    2**1
                      CONTENTS, ALLOC, LOAD, READONLY, CODE
                      ……
_START:
/* initialize stack pointer : move unline from GPR upline*/
         ldi _stack-8,%r8
c0700140:     1f 70          leri     0x301F          (0xFFFFF01F)
c0700142:     fe 7f          leri     0x3FFE          (0xFC07FFFE)
c0700144:     08 a8          ldi      0xC07FFFE8  %R8

                    ……

         jal _main
c0700154:     00 40          leri     0x0     (0x0)
c0700156:     00 40          leri     0x0     (0x0)
c0700158:     7f df          jal      c0700258 <_main>

c070015a <___main_init>:
#include <stdio.h>
__main_init(){};
c070015a:     20 b4          push     %lr
c070015c:     60 b0          push     %R5, %R6
c070015e:     d6 e1          lea      ( %SP    ) %R6

c0700160 <.LM2>:
c0700160:     c6 e1          lea      ( %R6     ) %SP
c0700162:     60 b1          pop      %R5, %R6
c0700164:     40 b5          pop      %pc

c0700166 <_Sub_Function>:
typedef struct {
int sum;
int car;
char str[7];
} TOTAL;
TOTAL Sub_Function(char one, int two, int three, TOTAL b)
{
c0700166:     20 b4          push     %lr
c0700168:     67 b0          push     %R0, %R1, %R2, %R5, %R6
c070016a:     d6 e1          lea      ( %SP    ) %R6
```

```
c070016c:   f7 b6          lea    ( %SP  0xFFFFFFDC   ) %SP
c070016e:   18 e4          lea    ( %R8     ) %R1
c0700170:   89 e4          lea    ( %R9     ) %R8
c0700172:   b6 02          ld     ( %R6   + 0x2C ) %R2
c0700174:   fd 7f          leri   0x3FFD        (0xFFFFFFFD)
c0700176:   36 12          st     %R2    , ( %R6          +
0xFFFFFFDC )
c0700178:   04 40          leri   0x4     (0x4)
c070017a:   06 38          stb    %R8    , ( %R6          + 0x20 )
c070017c:   fd 7f          leri   0x3FFD        (0xFFFFFFFD)
c070017e:   36 09          ld     ( %R6   + 0xFFFFFFDC ) %R9
c0700180:   09 08          ld     ( %R9   + 0x0 ) %R8
c0700182:   ff 7f          leri   0x3FFF        (0xFFFFFFFF)
c0700184:   06 18          st     %R8    , ( %R6          + 0xFFFFFFF0 )
c0700186:   fd 7f          leri   0x3FFD        (0xFFFFFFFD)
c0700188:   36 02          ld     ( %R6   + 0xFFFFFFDC ) %R2
c070018a:   12 08          ld     ( %R2   + 0x4 ) %R8
c070018c:   ff 7f          leri   0x3FFF        (0xFFFFFFFF)
c070018e:   16 18          st     %R8    , ( %R6          + 0xFFFFFFF4 )
c0700190:   fd 7f          leri   0x3FFD        (0xFFFFFFFD)
c0700192:   36 09          ld     ( %R6   + 0xFFFFFFDC ) %R9
c0700194:   29 08          ld     ( %R9   + 0x8 ) %R8
c0700196:   ff 7f          leri   0x3FFF        (0xFFFFFFFF)
c0700198:   26 18          st     %R8    , ( %R6          + 0xFFFFFFF8 )
c070019a:   fd 7f          leri   0x3FFD        (0xFFFFFFFD)
c070019c:   36 02          ld     ( %R6   + 0xFFFFFFDC ) %R2
c070019e:   32 08          ld     ( %R2   + 0xC ) %R8
c07001a0:   ff 7f          leri   0x3FFF        (0xFFFFFFFF)
c07001a2:   36 18          st     %R8    , ( %R6          + 0xFFFFFFFC )
c07001a4:   86 e4          lea    ( %R6     ) %R8
c07001a6:   d0 c8          addq   0xfffffff0,      %R8
c07001a8:   fd 7f          leri   0x3FFD        (0xFFFFFFFD)
c07001aa:   36 18          st     %R8    , ( %R6          +
0xFFFFFFDC )
```

```
c07001ac <.LM4>:
TOTAL a;
a.sum = two + one + b.sum;
                ……
c070023a <.LM13>:
return a;
```

```
c070023a:   96 e4          lea    ( %R6     ) %R9
c070023c:   fe 7f          leri   0x3FFE        (0xFFFFFFFE)
c070023e:   90 b8          add    0xFFFFFFE0  %R9
c0700240:   09 08          ld     ( %R9   + 0x0 ) %R8
```

```
c0700242:    01 18          st      %R8   , ( %R1          + 0x0 )
c0700244:    19 08          ld      ( %R9   + 0x4 ) %R8
c0700246:    11 18          st      %R8   , ( %R1          + 0x4 )
c0700248:    29 08          ld      ( %R9   + 0x8 ) %R8
c070024a:    21 18          st      %R8   , ( %R1          + 0x8 )
c070024c:    39 08          ld      ( %R9   + 0xC ) %R8
c070024e:    31 18          st      %R8   , ( %R1          + 0xC )

c0700250 <.LM14>:
}
c0700250:    81 e4          lea     ( %R1    ) %R8
c0700252:    c6 e1          lea     ( %R6    ) %SP
c0700254:    67 b1                  pop   %R0, %R1, %R2, %R5, %R6
c0700256:    40 b5          pop     %pc

c0700258 <_main>:
int main(void)
{
c0700258:    20 b4          push    %lr
c070025a:    61 b0          push    %R0, %R5, %R6
c070025c:    d6 e1          lea     ( %SP   ) %R6
c070025e:    f3 b6          lea     ( %SP  0xFFFFFFCC   ) %SP

c0700260 <.LM16>:
char one;
int two, three;
TOTAL b;
one = 1;
c0700260:    96 e4          lea     ( %R6    ) %R9
c0700262:    df c9          addq    0xffffffff,       %R9
c0700264:    01 a8          ldi     0x1     %R8
c0700266:    09 38          stb     %R8   , ( %R9          + 0x0 )

c0700268 <.LM17>:
two = 2;
c0700268:    02 a8          ldi     0x2     %R8
c070026a:    ff 7f          leri    0x3FFF          (0xFFFFFFFF)
c070026c:    26 18          st      %R8   , ( %R6          + 0xFFFFFFF8 )

c070026e <.LM18>:
three = 3;
c070026e:    03 a8          ldi     0x3     %R8
c0700270:    ff 7f          leri    0x3FFF          (0xFFFFFFFF)
c0700272:    16 18          st      %R8   , ( %R6          + 0xFFFFFFF4 )
```

```
c0700274 <.LM19>:
b.sum = 15;
c0700274:   96 e4          lea     ( %R6    ) %R9
c0700276:   fe 7f          leri    0x3FFE         (0xFFFFFFFE)
c0700278:   94 b8          add     0xFFFFFFE4  %R9
c070027a:   0f a8          ldi     0xF     %R8
c070027c:   09 18          st      %R8    , ( %R9          + 0x0 )

c070027e <.LM20>:
b = Sub_Function(one, two, three, b);
c070027e:   96 e4          lea     ( %R6    ) %R9
c0700280:   fe 7f          leri    0x3FFE         (0xFFFFFFFE)
c0700282:   94 b8          add     0xFFFFFFE4  %R9
c0700284:   86 e4          lea     ( %R6    ) %R8
c0700286:   df c8          addq    0xffffffff,        %R8
c0700288:   08 20          ldb     ( %R8    + 0x0 ) %R0
c070028a:   ff 7f          leri    0x3FFF         (0xFFFFFFFF)
c070028c:   26 08          ld      ( %R6    + 0xFFFFFFF8 ) %R8
c070028e:   83 98          st      %R8    , ( %SP          + 0xC )
c0700290:   ff 7f          leri    0x3FFF         (0xFFFFFFFF)
c0700292:   16 08          ld      ( %R6    + 0xFFFFFFF4 ) %R8
c0700294:   84 98          st      %R8    , ( %SP          + 0x10 )
c0700296:   86 e4          lea     ( %R6    ) %R8
c0700298:   fe 7f          leri    0x3FFE         (0xFFFFFFFE)
c070029a:   84 b8          add     0xFFFFFFE4  %R8
c070029c:   85 98          st      %R8    , ( %SP          + 0x14 )
c070029e:   89 e4          lea     ( %R9    ) %R8
c07002a0:   90 e4          lea     ( %R0    ) %R9
c07002a2:   ff 41          leri    0x1FF  (0x1FF)
c07002a4:   ff 7f          leri    0x3FFF         (0x7FFFF)
c07002a6:   5f df          jal     c0700166 <_Sub_Function>

c07002a8 <.LM21>:
return b.sum;
c07002a8:   86 e4          lea     ( %R6    ) %R8
c07002aa:   fe 7f          leri    0x3FFE         (0xFFFFFFFE)
c07002ac:   84 b8          add     0xFFFFFFE4  %R8
c07002ae:   08 08          ld      ( %R8    + 0x0 ) %R8

c07002b0 <.LM22>:
}
c07002b0:   c6 e1          lea     ( %R6    ) %SP
c07002b2:   61 b1          pop     %R0, %R5, %R6
c07002b4:   40 b5          pop     %pc

c07002b6 <___ctors>:
```

```
...
Disassembly of section .data:
```

[Table 4-18] Disassemble result of [Table. 4-19]

In this case, the argument and return value is pointer when the data type is structure.

4.5 Special Purpose Register Handling

4.5.1 Program Counter (PC)

PC register has a VMA of currently executed instruction + 2. If an assembly language programmer wants to get the PC value, trick codes are needed shown below

Code list – Get PC value
jal .LPC0 /* Jump to .LPC0 and Link to next instruction. LR = current PC + 2 */ jmp .LPC1 /* After returns dummy function, jump to .LPC1 */ .LPC0 : jplr /* Just returns */ .LPIC1 : push %lr /* Store the LR value to memory */ pop %R7 /* We can get stored PC value */ add 8, %R7 /* R7 register has current program counter, Recalculation!! */

4.5.2 Link Register (LR)

After executing JAL or JALR instruction, LR register has PC + 2.

Code list – Get LR value
push LR pop %R7 # R7 has LR value

4.5.3 Extension Register (ER)

ER register is updated when the EISC processor executes LERI instruction. The ER value is only valid when the SR register's E-flag field is set. Developer need not handling the ER register.

4.5.4 Multiply Result Register (MH, ML)

A result of multiplication or MAC is stored to MH/ML register.

Code list – Handling multiply result register
MTML %r0 /* Assign to %ML register with a vault %R0 */ MTMH %r1 /* Assign to %MH register with a vault %R1 */ MFML %r2 /* Assign to %R2 register with a vault %ML */ MFMH %r3 /* Assign to %R3 register with a vault %MH */
MUL %R0, %R1 MFML %R1 /* %R1 has a result value from %R0 * %R1*/

4.5.5 Count Register (CR0, CR1)

There are two special registers to support auto-increment in AE32000 architecture. The registers CR0, CR1 can be controlled by MTCR, MFCR instruction. MTCR instruction sets to the registers and MFCR instruction gets a value from the registers.

In AE32000 architecture has several auto-increment mode. You should refer to h/w reference manual to get more information. An example shown below is a code list for using auto-increment register in normal mode and that is a part of 'memmove' function in libc

Code list – Auto-increment load and store

```
        push %r7      # Store R7
        ldi 0,%r7     # Auto-increment mode setting
        mtcr0 %r7     #CR0 Set
        mtcr1 %r7     #CR1 Set
.L5:
        ldau 1, %r1, %r8      #Auto-increment Load    %r8 <- %r1 and CR1 = CR1 + 4
        stau 0, %r8, %r0      #Auto-increment Store   %r0 <- %r8 and CR0 = CR0 + 4
        ldau 1, %r1, %r8      #Auto-increment Load    %r8 <- %r1 and CR1 = CR1 + 4
        stau 0, %r8, %r0      #Auto-increment Store   %r0 <- %r8 and CR0 = CR0 + 4
        ldau 1, %r1, %r8      #Auto-increment Load    %r8 <- %r1 and CR1 = CR1 + 4
        stau 0, %r8, %r0      #Auto-increment Store   %r0 <- %r8 and CR0 = CR0 + 4
        ldau 1, %r1, %r8      #Auto-increment Load    %r8 <- %r1 and CR1 = CR1 + 4
        stau 0, %r8, %r0      #Auto-increment Store   %r0 <- %r8 and CR0 = CR0 + 4
        addq -16,%r9
        cmpq   15, %r9
        jhi .L5
        mfcr0 %r7     #CR0 load
        add %r7,%r0   # recover R0
        mfcr1 %r7     #CR1 load
        add %r7,%r1   # recover R1
        pop %r7       # recover R7
```

4.6 Assembly code programming syntax

4.6.1 Memory operation

① Load / Store with general registers

Opcode	Description
ld	Load 32-bit data form memory
lds	Load signed 16-bit data form memory (sign extend)
ldb	Load signed 8-bit data form memory (sign extend)
ldsu	Load unsigned 16-bit data form memory
ldbu	Load unsigned 8-bit data form memory
st	Store 32-bit data to memory
sts	Store 16-bit data to memory
stb	Store 8-bit data to memory

- There are 4 types operands possible.

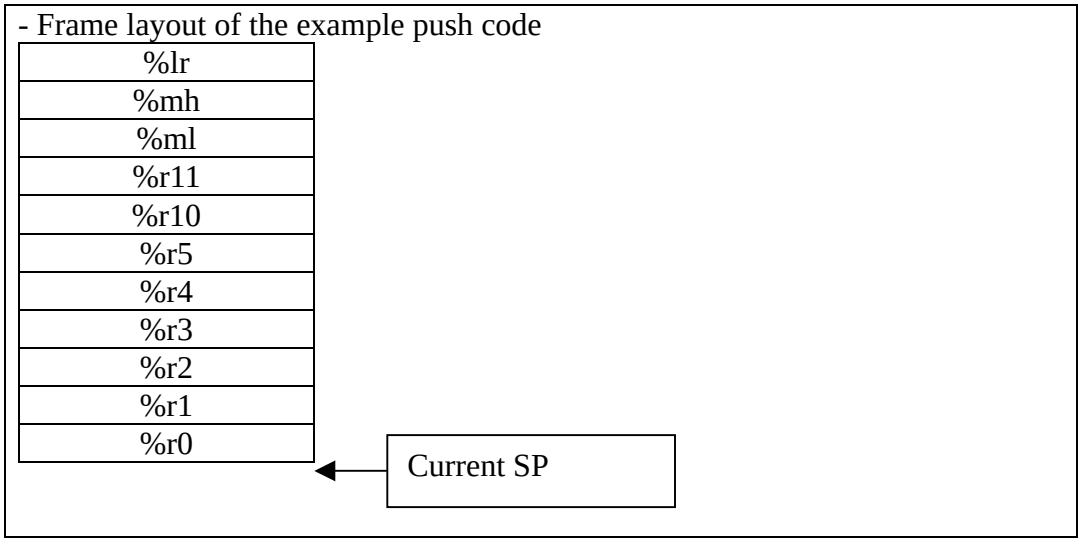
Code list – Usage load / store operation when base register is general register
opcode (%rt, x), %rd /* x is constant */ opcode (%rt), %rd opcode (x), %rd /* x is constant */ opcode (.label), %rd

② Load / Store with stack pointer register as a base

Code list – Usage load / store operation when base register is SP
opcode (%SP, x), %rd /* x is constant */ opcode (%SP), %rd

③ Push / Pop

Code list – Usage push / pop operation	
push %ml, %mh, %lr push %r10, %r11 push %r0 - %r5	pop %r0 - %r5 push %r10, %r11 push %ml, %mh, %pc



4.6.2 Arithmetic / Logical operation

Opcode	Description
add	Addition
addq	Addition with small immediate value (-16 ~ 15)
adc	Addition with carry
sub	Subtraction
sbc	Subtraction with carry
mul	Multiplication
mulu	Unsigned multimplication
and	Logical AND
or	Logical OR
xor	Logical XOR
asr	Arithmetic shift right
lsr	Logical shirt right
asl	Arithmetic shift left
ssl	Static shift left
cmp	Comparison
cmpq	Compare with small immediate value (-16 ~ 15)
extb	Sign extend byte
exts	Sign extend short
cvb	Convert to byte
cvs	Convert to short
neg	Negation
not	Logical NOT

- Arithmetic / Logical operations have 3 types.

Code list – Usage arithmetic/logical operation
opcode (%rt), %rd opcode (x), %rd /* x is constant */ /* addq, cmpq */ opcode %rd /* only for extb, exts, cvb, cvs, neg, not */

4.6.3 Branch operation

Opcode	Description
jnv	Jump on not overflow
jv	Jump on overflow
jp	Jump on positive
jm	Jump on minus
jnz	Jump on not zero
jz	Jump on zero
jnc	Jump on not carry
jc	Jump on carry
jgt	Jump on greater than
jlt	Jump on less than
jge	Jump on greater or equal
jle	Jump on less or equal
jhi	Jump on unsigned higher
jls	Jump on unsigned lover or equal
jmp	Always jump
jal	Jump and link
jr	Register indirect jump
jalr	Register indirect jump and link
jplr	Jump to link register

Code list – Usage branch operation
opcode x /* x is constant or symbol */ opcode label opcode %rd /* only for extb, exts, cvb, cvs, neg, not */ jplr /* jplr instruction needs not operands */

4.6.4 Move operation

Opcode	Description
lea	Move from/to stack pointer with offset
mov	Register move
mfcr0	Set general register from cr0 register

mfcr1	Set general register from cr1 register
mtcr0	Set cr0 register from general register
mtcr1	Set cr1 register from general register
mfmh	Set general register from mh register
mfml	Set general register from ml register
mtmh	Set mh register from general register
mtml	Set ml register from general register
ldi	Immediate constant move to general register

Code list – move operation

```
lea (%SP, x), %Rd          /* x is constant */
lea (%rt, x), %SP          /* x is constant */
mov %rt, %rd
mfcr0 %rd
mfcr1 %rd
mtcr0 %rt
mtcr1 %rt
mfmh %rd
mfml %rd
mtmh %rt
mtmh %rt
ldi x, %rd              /* x is constant */
```

4.6.5 DSP operation

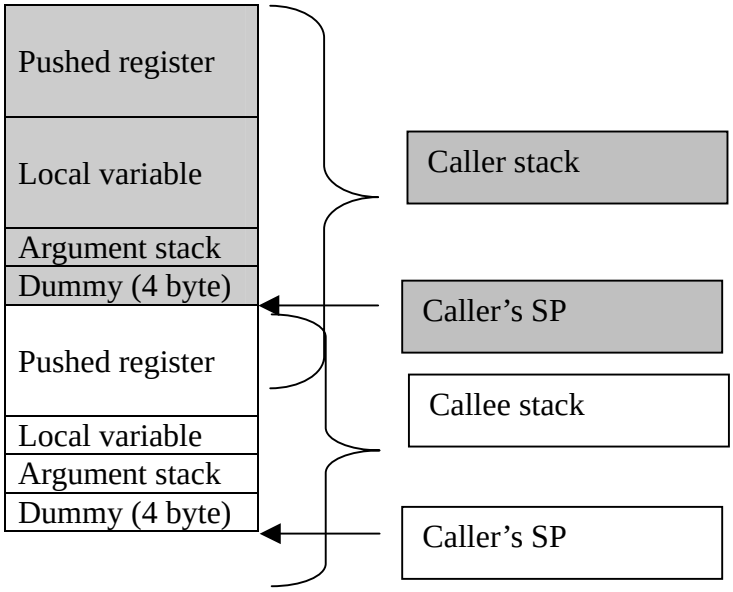
Opcode	Description
mac	Multiply and add operation

Code list – mac operation

```
mac %rt %rs
/*
mh:ml = mh:ml + rt * rs
*/
```

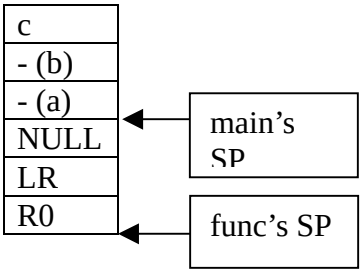
4.7 ABI (Application Binary Interface)

4.7.1 Frame layout



- A function's frame size is calculated by a formula
  (# pushed register * 4 + local variable size + # max argument * 4 + 4) byte.
- Pushed register is a register, which are stored to stack by caller. The registers are used in callee.
  - Compiler determines a local variable size at compile time. If developer compiles a source with –O0 optimization option, the local variable size is a summation of size of local variables. In order way, if developer compiles with –O2 option, compiler enters register allocation routine and the local variable size is a summation of spilled register size.
  - Max argument stack size is maximum size of summation callee's argument size.
  - AE32000 compiler uses %R8, %R9 register as argument register. If an argument is structure data type, the compiler passes to callee a pointer of the structure.

C source code	Compiler output (-O2)	Description

void main()
{
  int i1 = 1;
  int i2 = 2;
  int i3 = 3;
  func(i1, i2, i3);
}
 	
int func
(int a, int b, int c)
{
  return a+b+c;
}
	_main:
	  push %lr
	  lea (%sp,-16),%sp
	  ldi 3,%r8
	  st %r8,(%sp,12)
	  ldi 1,%r8
	  ldi 2,%r9
	  jal _func
	  lea (%sp,16),%sp
	  pop %pc
	  .size _main, .-_main
	  .align 1
	  .global _func
	  .type _func, @function
	_func:
	  push %lr
	  push %R0
	  ld (%sp,20),%r0
	  add %r9,%r8
	  add %r8,%r0
	  mov %r0,%r8
	  pop %R0
	  pop %pc
		A size of main's stack is 4(pushed LR) +16 = 20bytes. There is no area for local variables and the size 16 indicates max argument size + 4(dummy).
		
		A function 'func' has 8 bytes stack size that is only for pushed register. In callee side, to access third argument 'c', the '_func' uses an instruction "ld (%sp, 20), %r0". The value '20' is 8+ 4*3 and a frame layout is shown below.
		
		 

4.8 built-in functions

4.8.1 void * __builtin_return_address(unsigned int LEVEL)

- To get a return address of current executed function.

Code list – Example code for __builtin_return_address

```
int main()
{
        printf("Return address :: [%x]\n", __builtin_return_address(0) );
        return 0;
}
```

result :

```
Return address :: [1d8]
```

4.8.2 void * __builtin_frame_address(unsigned int LEVEL)

- To get a frame address of current executed function.

Code list – Example code for __builtin_frame_address

```
int main()
{
        printf("Return address :: [%x]\n", __builtin_frame_address(0) );
        return 0;
}
```

result :

```
Frame address :: [c07fffdc]
```

4.8.3 int __builtin_types_compatible_p(TYPE1, TYPE2)

- Check a compatible between TYPE1 and TYPE2. If the two argument are different, the function returns 0 otherwise 1.

Code list – Example code for __builtin_types_compatible_p

```
int main()
{
        printf("TYPE Compatible :: [%d]\n", __builtin_types_compatible_p(int , int) );
        return 0;
}
```

result :

```
TYPE Compatible [1]
```

4.8.4 int __builtin_constant_p(EXP)

- Check the EXP is constant or not.

Code list – Example code for __builtin_constant_p

```
int main()
{
        int a=1, b=2;
        printf("Constant :: [%d]\n", __builtin_constant_p(a, b) );
        return 0;
}
```

result :

```
Constant [1]
```

4.8.5 double __builtin_huge_val(void)

- Returns positive infinity as double data type.

4.8.6 float __builtin_huge_valf(void)

- Returns positive infinity as float data type.

4.8.7 Additional built-in functions

- In addition, AE32000 compiler supports ISC C, ISO C99, ISO C90 built-in functions.

4.8.8 ISO C mode

_exit, alloca, bcmp, bzero, dcgettext, dgettext, dremf, dreml, drem, exp10f, exp10l, exp10, ffsll, ffsl, ffs, fprintf_unlocked, fputs_unlocked, gammaf, gammal, gamma, gettext, index, j0f, j0l, j0, j1f, j1l, j1, jnf, jnl, jn, mempcpy, pow10f, pow10l, pow10, printf_unlocked, rindex, scalbf, scalbl, scalb, significandf, significandl, significand, sincosf, sincosl, sincos, stpcpy, strdup, strfmon, y0f, y0l, y0, y1f, y1l, y1, ynf, ynl

4.8.9 ISO C99 functions

Exit, acoshf, acoshl, acosh, asinhf, asinhl, asinh, atanhf, atanhl, atanh, cabsf, cabsl, cabs, cacosf, cacoshf, cacoshl, cacosh, cacosl, cacos, cargf, cargl, carg, casinf, casinhf, casinhl, casinh, casinl, casin, catanf, catanhf, catanhl, catanh, catanl, catan, cbrtf, cbrtl, cbrt, ccosf, ccoshf, ccoshl, ccosh, ccosl, ccos, cexpf, cexpl, cexp, cimagf, cimagl, cimag, conjf, conjl, conj, copysignf, copysignl, copysign, cpowf, cpowl, cpow, cprojf, cprojl, cproj, crealf, creall, creal, csinf, csinhf, csinhl, csinh, csinl, csin, csqrtf, csqrtl, csqrt, ctanf, ctanhf, ctanhl, ctanh,

ctanl, ctan, erfcf, erfcl, erfc, erff, erfl, erf, exp2f, exp2l, exp2, expm1f, expm1l, expm1, fdimf, fdiml, fdim, fmaf, fmal, fmaxf, fmaxl, fmax, fma, fminf, fminl, fmin, hypotf, hypotl, hypot, ilogbf, ilogbl, ilogb, imaxabs, lgammaf, lgammal, lgamma, llabs, llrintf, llrintl, llrint, llroundf, llroundl, llround, log1pf, log1pl, log1p, log2f, log2l, log2, logbf, logbl, logb, lrintf, lrintl, lrint, lroundf, lroundl, lround, nearbyintf, nearbyintl, nearbyint, nextafterf, nextafterl, nextafter, nexttowardf, nexttowardl, nexttoward, remainderf, remainderl, remainder, remquof, remquol, remquo, rintf, rintl, rint, roundf, roundl, round, scalblnf, scalblnl, scalbln, scalbnf, scalbnl, scalbn, snprintf, tgammaf, tgammal, tgamma, truncf, truncl, trunc, vfscanf, vscanf, vsnprintf and vsscanf
acosf, acosl, asinf, asinl, atan2f, atan2l, atanf, atanl, ceilf, ceill, cosf, coshf, coshl, cosl, expf, expl, fabsf, fabsl, floorf, floorl, fmodf, fmodl, frexpf, frexpl, ldexpf, ldexpl, log10f, log10l, logf, logl, modfl, modf, powf, powl, sinf, sinhf, sinhl, sinl, sqrtf, sqrtl, tanf, tanhf, tanhl and tanl

4.8.10 ISO C90 functions

abort, abs, acos, asin, atan2, atan, calloc, ceil, cosh, cos, exit, exp, fabs, floor, fmod, fprintf, fputs, frexp, fscanf, labs, ldexp, log10, log, malloc, memcmp, memcpy, memset, modf, pow, printf, putchar, puts, scanf, sinh, sin, snprintf, sprintf, sqrt, sscanf, strcat, strchr, strcmp, strcpy, strcspn, strlen, strncat, strncmp, strncpy, strpbrk, strrchr, strspn, strstr, tanh, tan, vfprintf, vprintf and vsprintf

Chapter 5
Exception Handling

Exception handling is a programming language construct or computer hardware mechanism designed to handle the occurrence of some condition that changes the normal flow of execution. EISC processor supports the exception handling hardware and software. In this chapter, how the mechanism works on software side and shows programming method.

5.1 Exception of processor

Exception is a signal to handling an asynchronous with CPU exception such as reset and interrupt. There are two kinds of exception signals. First is for special handling to operate system. For example, if developer wants to see an output screen certain application, the application interrupts a CPU and print screen the output. Another is for application needs a special handling. For example, to debug an application, GDB sends a break signal to system. The ESIC system receives the signal and enters break point management routine. All kinds of exception have a different handling routine. This chapter will explain the exceptions later.

It is sure that there is priority among the exceptions. Even if the system operates an exception handling and enters a new signal with higher priority, the system stops the operation and works for the new signal. On the other hands, if the new exception has lower priority, the exception must wait until the system finishes a current exception handling.

EISC system has three modes that are User Mode, Supervisor Mode and OSI Mode. If exception occurs, CPU turns to Supervisor mode and executes exception handling. However, if OSI exception occurs, CPU executes OSI program on the OSI mode. The CPU mode is determined by SR (Status Register), changed automatically when receives exception signal. After exception handling, the SR register will be POP and recovers CPU mode. Several EISC processors does not have the OSI mode but if exist, there is ISP (OSI Stack Pointer register).

5.2 Exceptions

5.2.1 Reset

This exception will occur when a request for the processor initialization is made. The external signal to the core processor will trigger this exception. If the processor gets this exception, it begins initialization processor. All the values of the general-purpose registers will be initialized to either 'unknown' value or '0' (As this value can be different according to the processor model, please refer to the technical reference manual). The value of the PC will be set to that of the reset vector and the SR will be set to zero. The value of the LR, ER, MH, ML, SSP, ISP and USP are set together 'unknown' or zero based on the implementation.

5.2.2 External Hardware Interrupt

This is the external interrupt from the external hardware module. After PUSH current context into a stack, the execution will jump to the interrupt handler.
Among the current context, PC and SR value will be PUSHed into the supervisor stack and other registers that need PUSH operations should be managed by the interrupt handler.

Auto-vectored Interrupt : Interrupt vector is specified. When an interrupt occurs, jump to the pre-assigned interrupt vector.
Vectored Interrupt : Interrupt number should be received from the external interrupt controller to the interrupt vector.

5.2.3 Software Interrupt

This is the interrupt occurring from the software. When we want to access the resources of the supervisor domain from the user mode, this interrupt can be used. Generally, this interrupt is used for the system call. The current context of the processor will be PUSHed and the corresponding interrupt vector with the specified interrupt number will be accessed.

5.2.4 Non-Maskable Interrupt

The Non-Maskable Interrupt (NMI) is an interrupt whose masking(denial of the interrupt request) is not allowed in the interrupt controller. In general, it is used when parity errors are detected. But in AE32000, as the system coprocessor invokes an interrupt in such situation, (refer to the system coprocessor interrupt) it is connected to the specific external interrupt. In AE32000, the NMI is used as the interrupt that is input directly without going through the interrupt controller.

5.2.5 System Coprocessor Interrupt

This interrupt occurs during the memory access in the system coprocessor. For example, when a user tries to access the resources of the supervisor domain from the user mode or when

the destination address is invalid or when a parity error is detected during the memory read/write process, the system coprocessor interrupts will occur.

5.2.6 Coprocessor Interrupt

This is the interrupt that occurs during the access of the coprocessor, and the interrupt can also be triggered by polling the operation result (status bit) of the operating co-processor. The former case happens when a co-processor that is dedicated only for the supervisor receives access requests from the user mode.

5.2.7 Breakpoint & Watchpoint Interrupt

These interrupts are provided to support OSI debugging. When the conditions specified in the breakpoint and watchpoint of the OSI module are satisfied, these interrupts will be generated and the processor mode will be switched to the OSI mode.

5.2.8 Bus Error & Double Fault

If irrecoverable errors are detected in the instruction bus or data bus, the Bus error occurs. If an error occurs during the interrupt handler fetch or push operation, as we cannot proceed to the interrupt handler any more, double fault will be generated. This can happen when there are some problem in the supervisor stack of if the vector base register that modifies the location of the interrupt handler was not configured properly.

5.2.9 Undefined Instruction Exception

This occurs when an undefined instruction is detected. In this case, either an error occurs or it is replaced with the NOP instruction based on the version of the processor.

5.2.10 Unimplemented Instruction Exception

This is the instruction that are defined in the ISA, but not implemented yet in current version. In this case, we can get the value by using the software emulation through the interrupt. They are used to increase compatibility.

5.3 Interrupt Vector Table

5.3.1 Introduction

To handle the exceptions, the processor should access the interrupt handler routines and the addresses of them are stored in the interrupt vector table. The interrupt vector table of the AE32000 is located at the address 0x0 by default and the interrupts are defined as shown in the [Figure 5-1]. The shaded interrupts cannot be moved to the vector base register.

Word Size	
Reset	0x00000000
NMI	0x00000004
INT (Auto)	0x00000008
Double Fault	0x0000000C
Bus Error	0x00000010
Reserved	0x00000004 0x0000001C
CP0 Exception	0x00000020
CP1 Exception	0x00000024
CP2 Exception	0x00000028
CP3 Exception	0x0000002C
RESET (OSI)	0x00000030
OSI (OSI)	0x00000034
Undefined Instruction	0x00000038
Unimplemented Instruction	0x0000003C
SWI	0x00000040 0x0000007C
INT	0x00000080 0x000004FC
Reset (OSIROM)	0xFFFF0000
OSI (OSIROM)	0xFFFF0004

[Figure 5-1] Interrupt vector table

[Figure. 5-1] shows an array of the pointers of the interrupt handler functions. In AE32000 system, it is located at the address 0x0. The interrupt vector table can be relocated if necessary but the shaded interrupts cannot be relocated.

As the interrupt vector should be located at the address 0x0, when using the C compiler, we should specify the memory layout in the linker script such that the interrupt vector is located at 0x0.

```
SECTIONS
  {
  /* Read-only sections, merged into text segment: */
  . = 0x0;
  .text :
  {
*(.vects)        #indicates a position of interrupt vector
```

[Table 5-1] Linker script that locates the interrupt vector at the address 0x0

The '.vects' above is a symbol that indicates the section. The linker arranges the memory base on these section symbols.

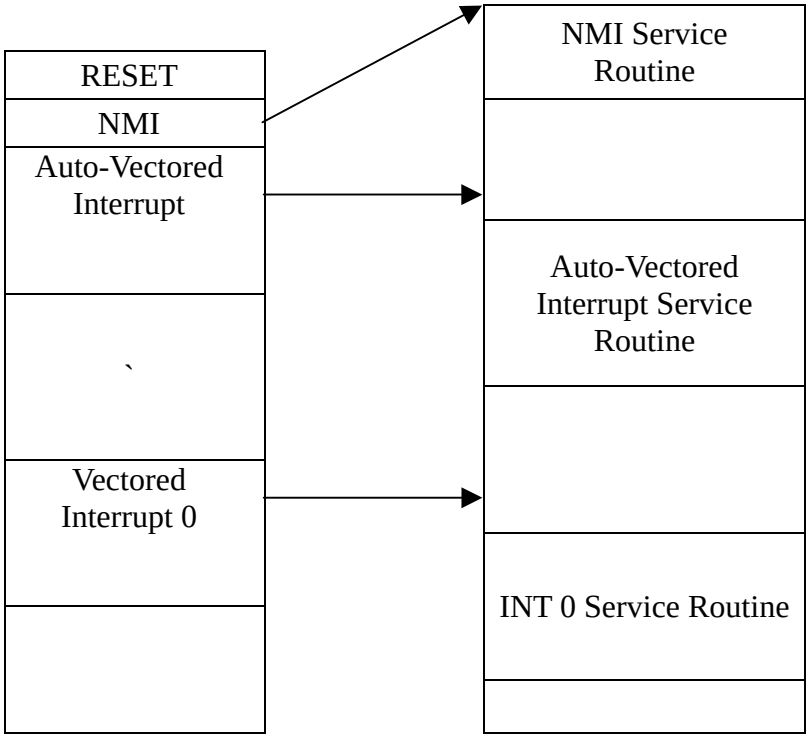
```
typedef void (*fp)(void);
const fp HardwareVector[] __attribute__((section (".vects")))= {
      start, /* V00 : Reset Vector */
      nmi, /* V01 : NMI Vector */
      auto_int, /* V02 : Interrupt Auto Vector */
      dfault_int, /* V03 : Double fault Vector */
      berr_int, /* V04 : Bus Error Exception */
      …
   };
```

[Table 5-2] Definition of the interrupt vector table

As we can see from the [Table 5-3] above, the vector table does not have any parameters and is composed of the function pointers that don’t have any return values. As the attribute of the table, it set the section symbol to '.vects' so that the linker can recognize it.

5.3.2 Exception Vector and Exception Service Routine

[Figure 5-2] shows a relationship between exception vector and exception service routine. Exception vector is a pointer of start address that points to exception service routine. The EISC processor executes the exception service routine after the processor reads the exception vector and assigns to PC register.



[Figure 5-2] A relations between exception vector and exception service routine.

The exception service routine cannot have a input/output. It means the service routine has a void prototype. Therefore, the exception service routine has a form like as [Table 5-3].

```
#pragma interrupt
void INTERRUPT_FUNCTION(void)
{
        … ..
        -- Exception service
        … ..
}
```

[Table 5-3] Simple exception service routine.

There are two kind of interrupt functions. These are distinguished by pragma, one is #pragma interrupt and #pragma interrupt_fast. #pragma directive announces to compiler that a function is interrupt function, and compiler inserts additional codes to prologue and epilogue in order to store and roll-back current state.

At the first of interrupt function, processor pushes all of general registers and CR0, CR1, ML, MH, ER and LR registers. It means the processor stores current states before execute interrupt handles. On the other hands, at finish time the interrupt, all of pushed registers are poped in order to recover state.

A interrupt function with #interrupt_fast pragma does not store the general registers. If the interrupt function does not use general register, developer had better use the #interrupt_fast to get higher performance.

```
#pragma interrupt
void INTERRUPT_FUNCTION(void)
{
   asm ( “
      push %r0-%r7
      push %r8-%r15
      push %lr, %er
   “);
   … ..
   -- Exception service
   … ..
asm (“
pop %lr,%er
pop %r8-%r15
pop %r0-%r7
“);
}
```

[Table 5-4] C source and compiled assembly code of an interrupt function

Software interrupt receives an argument from register %R8. Due to this reason, before compiler calls SWI, compiler sets a value to %R8. SWI handler reads the arguments and enters service routine.

```
#pragma interrupt
void SWI_FUNCTION(void)
{
int input1, input2;
int output1, output2;
asm ( “
push %r0-%r2,%r5-%r7
push %r8-%r15
```

```
push %lr, %er
“);
asm ( “ st %%r0, %0”::”g”(input1) );
asm ( “ st %%r1, %0”::”g”(input2) );
… ..
-- Exception service
… ..
asm ( “ ld %0, %%r3”::”g”(output1) );
asm ( “ ld %0, %%r4”::”g”(output2) );
asm (“
pop %lr,%er
pop %r8-%r15
pop %r0-%r2,%r5-%r7
“);
}
```

[Table 5-5] An example of SWI

```
Const fp HardwareVector[] __attribute__((section (".vects")))= {
/* Reset Vector */
_START, /* V00 : Reset Vector */
nmi_serv, /* V01 : NMI Vector */
auto_int_serv, /* V02 : Interrupt Auto Vector */
NOTUSEDISR, /* V03 : Reserved */
NOTUSEDISR, /* V04 : Reserved */
NOTUSEDISR, /* V05 : Reserved */
NOTUSEDISR, /* V06 : Reserved */
NOTUSEDISR, /* V07 : Reserved */
… … .
}
#pragma interrupt
void nmi_serv(void)
{
asm ( “
push %r0-%r7
push %r8-%r15
push %lr, %er
“);
… ..
-- NMI exception service
… ..
asm (“
pop %lr,%er
pop %r8-%r15
pop %r0-%r7
```

```
“);
}
#pragma interrupt
void auto_int_serv (void)
{
asm ( “
push %r0-%r7
push %r8-%r15
push %lr, %er
“);
… ..
--Auto-vectored interrupt service
… ..
asm (“
pop %lr,%er
pop %r8-%r15
pop %r0-%r7
“);
}
```

[Table 5-6] Examples of interrupt suvice functions.

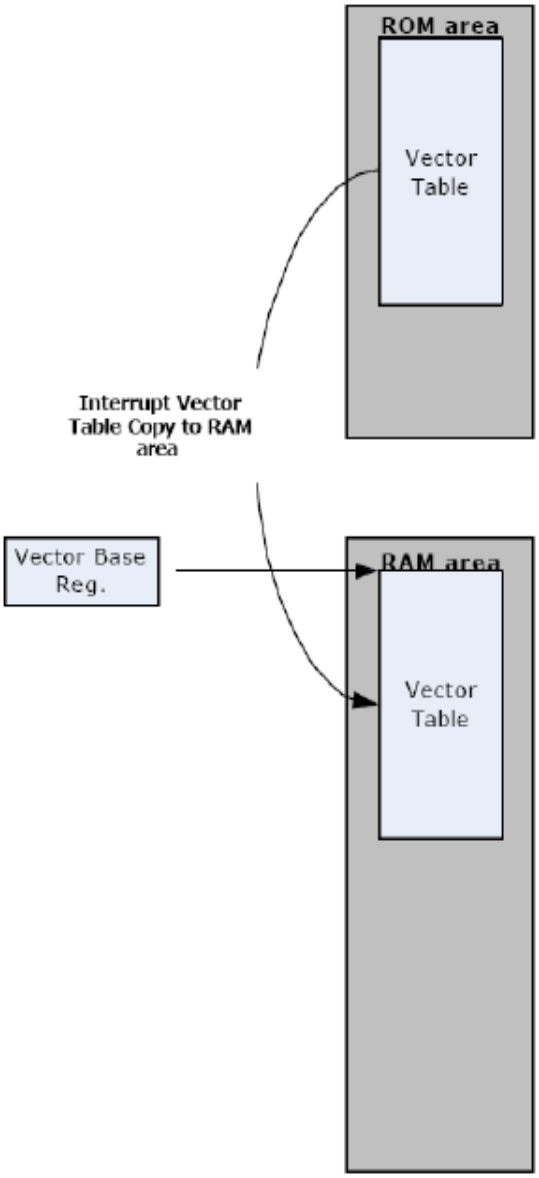
5.4 Vector Base Interrupt

5.4.1 Description

The vector base specifies the location of the interrupt vector table. In general, the interrupt vector table is located at the address 0x0 and some of the interrupt vectors that perform inexact exception processing cannot be relocated (there are marked as the shaded region in the [Figure 5-1] above.). The vector base is stored in the vector base register of the system co-processor and after this value is specified, most interrupts will read the vector address from the modified interrupt vector table.

Vector base register can change the vector table access from ROM region to the RAM region to shorten the execution time and enable the interrupt hooking.

The specific vector table region in the vector base should be located inside the TLB(Translate Look-aside Buffer) if we are using the virtual memory. The reason is that as the AE32000adopts software-managed TLB, if a TLB miss exception is thrown during the vector table access, we might get unnecessary double fault.



[Figure 5-3] Relocation of the interrupt vector table

When relocating the interrupt vector table, we can specify the location using the vector base register. If the vector base register is specified, when interrupt occurs, the vector table will be accessed based on the vector base register.

5.5 Exception Priority

The interrupts in the Lucifer processor have priorities. The internal priorities are shown below.

Priority	Exception	Abbreviated Name
1	Reset	RST
2	Bus Error	BERR
3	Double Fault	DF
4	Osi Exception	OSI
5	Co-processor Exception	CP
6	System Co-processor Exception	CP0
7	Non-Maskable Interrupt	NMI
8	Software Interrupt	SWI
9	Interrupt	INT
10	Halt	HALT
11	Undefined Instruction Exception	UDI
12	Unimplemented Instruction Exception	UII

[Table 5-7] Exceptions' priority

This priority values are used when selecting one of the interrupts that occur simultaneously and generally, the one with the earlier request time will have precedence.

5.6 Entering to the Exception Service Routine

EISC processor checks whether an occurred exception has higher priority than current exception. If the new exception has a higher priority, the EISC processor operates 4 steps shown below.

1. Stores SP and PC register value to stack in order to execute current exception after finish.
2. CPU mode changing by modifying the SR register.
3. Read an address of service routine according to the new exception and assigns to PC.
4. Execute service routine.

If the exception is OSI Exception, CPU mode is changed to OSI mode and a stack point is OSI stack(ISP). When the other exception occurs, saves the registers to supervisor stack with Supervisor Stack Pointer (SSP).

In addition, if the current exception is Halt, all of the exceptions are taken off HALT except SWI and Double Fault.

Exception	Action
RESET	**Clear HALT flag** **Clear %SR** **Move ID-CODE to %R0** **Get %PC from 00000000h**
NMI	**Clear HALT flag** **Push %SR by using %SSP** **Clear %SR.b15 , %SR.b14, %SR.b13, %SR.b11** **Push %PC** **GET %PC from 00000004h**
INT	**Clear HALT flag** **Push %SR by Using %SSP** **Clear %SR.b15 , %SR.b13, %SR.b11** **Push %PC** **If (%SR.b12 == 0) Get %PC from 0000 0008** **Get Interrupt vector V at interrupt acknowledge cycle** **GET %PC from (0000000Vh) * 4**
SWI	**Push %SR by using %SSP** **Clear %SR.b15 , %SR.b13, %SR.b11** **Push %PC** **Get %PC from (0000000Nh) * 4 + 00000040h**
Privilege Violation	**Push %SR by using %SSP** **Clear %SR.b15 , %SR.b13, %SR.b11** **Push %PC** **Get %PC from 0000000Ch**

Co-Processor Exception	**Clear HALT flag** **Push %SR by using %SSP** **Clear %SR.b15 , %SR.b13, %SR.b11** **Push %PC** **Get %PC from (0000000Nh) * 4 + 00000080h**
Double Fault Exception	**Disable cache and TLB if CP0** **Clear SR** **Get %PC from 00000010h**
Exception	Action

[Table 5-8] shows operations within each exceptions

5.7 Finishing Exception Service Routine

To return to old state after finishing the exception service, CPU should recover all the registers, which are stored in stack. A pop operation is opposite way of push. After the EISC system pops general registers and special registers, the exception service routine is finished and returns to a program before occurring exception. PC register has to be the latest popped register because the changing PC means the exception service is finished.

```
_auto_int:
push %lr,%er
push %r0-%r3 /* Pushes registers to use*/
… …
pop %r0-%r3
pop %lr,%er
pop %pc,%sr /* Returns to old process*/
/* end auto-vectored interrupt service routine */
```

[Table 5-9] Example of Auto-Vectored Interrupt Service Routine written in assembly language

```
#pragma interrupt
void auto_vector_int(void)
{
        asm (“
                push %lr,%er
                push %r0-%r8
                push %r9-%r15
        “);
        … … .
        Asm (“
                pop %r9-%r15
                pop %r0-%r8
                pop %lr,%er
        “);
}
```

[Table 5-10] Example of Auto-Vectored Interrupt Service Routine written in C language

If developer uses SWI, data communication can be occurred between an application and an exception service routine. Therefore, you have to check the ABI and implement interrupt function with assembly language.

5.8 Additional comment for Exception Handling

5.8.1 Reset

When a reset pin of CPU turns to enable, CPU stops all of executions and enters to the exception handling. In AE32000 architecture, the CPU clears status register (SR) and load reset vector according to whether an OSI module enable or not. If both OSIEN and OSIROM are enable, CPU load the reset vector from 0xFFFF_0000 and if OSIEN is enable but OSIROM is disable, CPU load the reset vector from 0x0000_0030.

5.8.2 NMI

NMI interrupt is used for I/O handling. If NMI occurs in AE32000 architecture, CPU stores the SR to stack with SSP(Supervisor Stack pointer) value. After then, AE32000 disables the NMI interrupt and clears extension flag and clears processor mode to enter the supervisor mode. Finally, the CPU stores the PC register to stack and loads NMI vector and executes it.

5.8.3 SWI

SWI is occurred by user application. The SWI instruction is detected; CPU changes a mode to Supervisor Mode. SR register is stored to stack with SSP value. After then AE32000 disables the NMI interrupt and clears extension flag and clears processor mode to enter the supervisor mode. Finally, the CPU stores the PC register to stack and loads SWI vectored and executes it.

5.8.4 Interrupt

If 12th-bit of SR register is reset, the interrupt works as Auto-Vectored Interrupt, and if set, the interrupt works as Vectored Interrupt. In the Auto-Vectored Interrupt case, CPU loads a exception vector from memory 8, however, Vectored interrupt case, CPU loads from external to CPU.

The mechanism to handling the interrupt is same as SWI but just different from the address of exception vector.

5.8.5 OSI Break Exception

OSI is a debugging module inside on chip. With OSI interrupt, developer can debug an implemented application on EISC system. There are 8 break points comparator. OSI break interrupt is occurred by external signal and compares two values between OSI breakpoint and PC or data address.

You want to get more information about OSI; you should refer to OSI Debug manual.

5.8.6 Co-Processor Exception

At the point Co-processor exception occurring, CPU gets an address according to the Co-processor number. If the EISC system shares the address bus and Co-processor bus, it can be complicate. So, the address bus is different with Co-processor number bus.

5.8.7 Bus Error Exception

Informing to CPU when error occurs on system bus. Thus, null pointer access, invalid data access can occurs this exception from memory controller. This exception cannot handling to execute right way, so the EISC system resets all process and must restart the system.
