advanced digital chips inc.

EISC Software Developer Guide

Extendable Instruction Set Computer

Ver 2.0

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

Change log

2008.07.23.

문서 최초 배포

목 차

Figure

Table

1 장

소개

이 장에서는 Advanced Digital Chips Inc.에서 개발한 EISC (Extendable Instruction Set Computer) 프로세서에 대한 소개와 EISC 컴파일러를 사용하여 코드를 개발하는 방법을 설명합니다. 여기에는 다음 단원이 포함되어 있습니다.

- EISC architecture 소개
- EISC Compiler tool chain
- 일반 프로그래밍 문제
- 임베디드 소프트웨어 개발

1.1 EISC architecture 소개

1.1.1 EISC overview

㈜에이디칩스(http://www.adc.co.kr)에서 개발한 EISC 프로세서는 내장형 응용 분야(embedded application)에 최적화된 명령어를 제공한다. EISC 프로세서는 메모리 접근에 직접적인 영향을 주는 프로그램의 크기를 줄이면서도 적절한 수의 레지스터를 제공함으로써 데이터 메모리 접근 빈도를 줄이는 방법을 사용하고 있다. 이러한 접근 방법은 형태상 축약 명령어를 사용하는 RISC의 접근 방법과 매우 유사하나 leri 라 불리는 즉치 값 확장 명령어를 이용하여 필요한 경우 명령어에 사용되는 즉치 값(immediate value)이나 변위(offset)를 자유롭게 확장할 수 있는 형태를 가질 수 있도록 함으로써 즉치 값 사용의 문제를 해결하였다는 장점을 지니고 있다.
또한 EISC 프로세서는 구조적으로 RISC와 유사하므로 하드웨어 형태가 비교적 단순하다는 장점을 지니고 있으며, 명령의 확장의 방식에 있어서 CISC와 유사점이 있으므로 CISC와 필적하는 코드 밀도를 보여준다는 장점을 지닌다.

1.1.2 Software Development Environment

EISC 프로세서 기반의 시스템에서 소프트웨어를 개발할 때 다음과 같은 개발 프로그램이 필요하다.

• IDE(Integrated Development Environment)
– EISC Studio : EISC 프로세서 기반의 시스템을 개발하기 위한 통합 개발 환경으로서, Microsoft™ Windows 환경에서만 사용 가능하다.

• GNU 기반의 Tool chain : GNU 기반의 컴파일러/디버거/시뮬레이터를 제공한다.
  ♦ Support platform : Linux, Microsoft™ Windows(Cygwin)
    – GNU C/C++ Compiler
    – GNU macro assembler, linker
    – GNU Debugger (GDB), Insight GUI debugger
    – GNU instruction set simulator
• ESCAsim: EISC System Cycle Accurate Simulator

1.1.3 이 문서 활용하기

이 가이드는 C/C++ 및 어셈블러의 기초적이고 기본적인 지식을 필요로 한다. 또한 EISC는 기본적으로 임베디드 시스템의 개발에 적합한 Architecture이므로 임베디드 시스템의 개발을 위해 어느 정도의 하드웨어적 상식을 갖추고 있어야 한다.
이 Software Developer Guide는 예제 프로그램을 통해 EISC Architecture의 Software Tool 사용법 및 프로그래밍에 기본적인 도움을 준다. 본 가이드의 효율적인 활용을 위해서 GNU GCC/Binutils/GDB Manual, 임베디드 시스템, Computer Architecture 등의 자료를 참고하기 바란다.

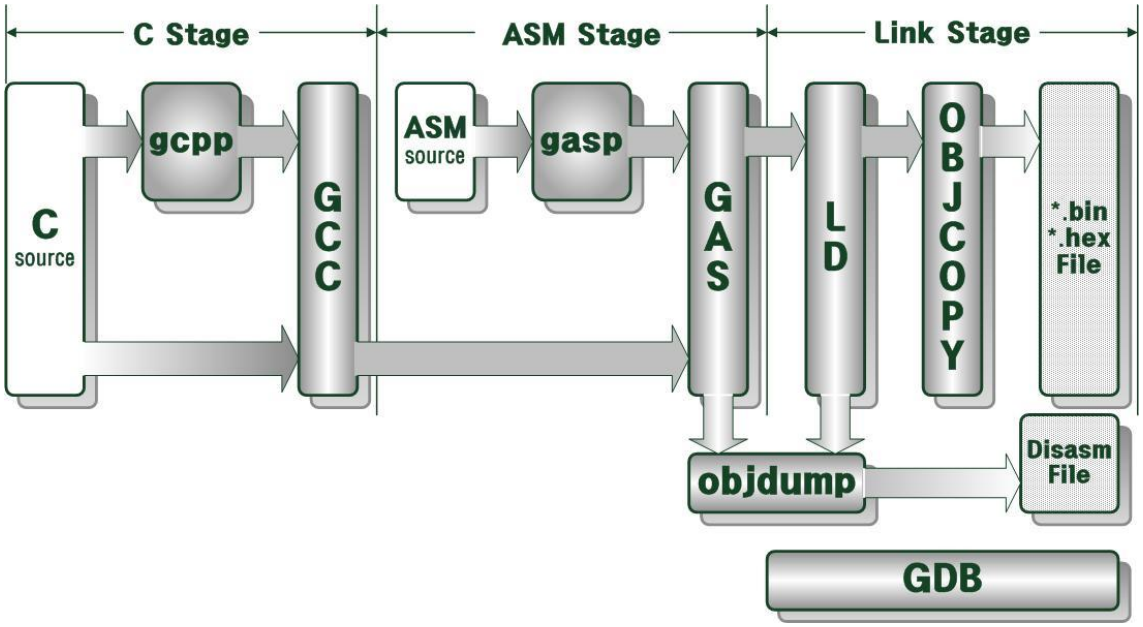
1.2 EISC Compiler tool chain

EISC 컴파일러는 기본적으로 GCC 의 사용법을 따르는 크로스 컴파일러 이다. GCC는 매우 뛰어난 성능과 안정성을 갖추고 있으며 GNU프로젝트의 일환으로 개발된 공개 C/C++ 언어 컴파일러이다. GCC에 대한 더욱 자세한 사용법은 http://gcc.gnu.org에서 구할 수 있다.

우선 Tool chain 이란 타겟 시스템의 소프트웨어 개발을 진행하기 위해 필요한 호스트 시스템의 크로스 컴파일 환경을 뜻한다.

소스 코드를 컴파일하고 빌드하여 binary 실행파일을 생성하는데 필요한 각종 유틸리티 및 라이브러리의 모음이다. Tool chain은 기본적으로 어셈블러, Linker, C/C++ 컴파일러, C/C++ 라이브러리 등으로 구성되어있다.

크로스 컴파일러(Cross Compiler)를 이해하기 전에 먼저 호스트 플랫폼(host platform)과 타겟 플랫폼(target platform)이라는 용어를 먼저 이해해야 한다. 타겟 플랫폼은 크로스 컴파일러가 생성하는 Object 코드가 실제로 수행되는 시스템을, 호스트 플랫폼은 크로스 컴파일러를 수행하는 시스템을 일컫는다. 예를 들어 사용자의 컴퓨터 상에서 EISC C/C++ 컴파일러를 이용하여 EISC용 Object 코드를 생성하고 그 결과 코드를 EISC에서 수행하였을 경우에, 사용자의 컴퓨터는 호스트 플랫폼이 되는 것이고, EISC는 타겟 플랫폼이 된다. 여기에서 크로스 컴파일러 라는 말이 생긴 것이다.



[그림 1-1] EISC 프로세서를 위한 C 컴파일러의 동작

[그림 1-1]은 컴파일러의 동작을 전체적으로 보여주고 있다. C 프로그램은 C 컴파일러에 의해서 어셈블리 소스 코드로 변환된다. 어셈블러는 어셈블리 소스를 Object 파일로 변환시키고, Linker에 의해서 여러 Object 파일들을 Link하여 최종적인 Binary 실행파일을 생성해 낸다

ECOMI 는 “EISC Compiler Installer”로 Microsoft™ Windows 용 EISC 컴파일러 인스톨 프로그램을 의미하며 각 프로세서에 맞는 EISC 컴파일러와 Cygwin을 설치하는 프로그램이다.

ECOMI를 사용해 EISC 컴파일러를 설치하는 경우 ECOMI 압축 해제 후 SETUP.EXE 파일을

실행시켜 설치할 수 있다. 기존 Cygwin 사용자의 경우 컴파일러만 추가로 설치하여 사용할 수 있다. 컴파일러는 cygwin이 설치된 경로의 ₩usr₩local에 설치되어 있다. 아래의 표는 ₩usr₩local폴더 안에 설치된 폴더에 대한 설명이다.

폴더 명	설명
bin	AE32000 compiler tool chain 실행 파일을 포함한다.
lib	libc, libm등 compiler가 실행파일을 만드는데 필요한 라이브러리를 포함한다.
man	Compiler tool chain 실행파일의 manual 이다.
include	컴파일 시 필요한 헤더파일들을 포함하고 있다.
ae32000-elf	AE32000 target specific 폴더이다.

1.3 일반 프로그래밍 문제

EISC 프로세서는 구조적으로 RISC 프로세서와 유사하다. 이 타입의 장치에서는 일반적으로 코드의 효율성을 높이는 여러 가지 프로그래밍 전략이 지원된다.
EISC 프로세서는 대부분의 RISC 프로세서와 마찬가지로 정렬된 데이터, 즉 int, long 형은 4의 배수인 주소에 있는 워드, short 형은 2의 배수인 주소에 있는 하프워드 그리고 char 형은 모든 주소에 있는 워드에 액세스하도록 설계되었다. 이 데이터는 해당 기본 크기 경계에 있어야만 한다.
EISC 컴파일러는 전역 변수는 이러한 기본 크기 경계에 따라 정렬하므로 LD 및 ST 명령어를 사용하여 이러한 항목에 효율적으로 액세스할 수 있다.

1.3.1 정렬되지 않은 포인터

코드 크기를 줄이고 성능을 향상시키기 위하여 한 타입에 대한 포인터를 해당 타입의 기본 정렬보다 작은 단위로 정렬할 수는 없다. 따라서 EISC 컴파일러에서는 기본적으로 표준 C 및 C++ 포인터가 메모리 내의 정렬된 워드를 가리키는 것으로 가정한다. 정렬되지 않은 데이터를 액세스하는 코드를 사용 시에는 주의가 요구된다.
예를 들어, 워드를 읽도록 사용된 int 형 포인터의 경우 EISC 컴파일러는 LD 명령어를 사용한다. 이 명령어는 주소가 4의 배수일 때, 즉 워드 경계에 있을 때 정상적으로 실행된다. 그러나 주소가 4의 배수가 아니면 코드가 주소 0x0006을 가리키는 포인터에서 데이터를 로드하는 경우 즉, 0x0006, 0x0007, 0x0008 및 0x0009에서 바이트의 내용을 로드하려 할 때 LD 명령어는 정렬되지 않은 워드를 실제로 로드하지 않고 Data Misalign Error를 발생한다.
따라서 기본 정렬 경계에 있지 않고 임의의 주소에 있을 수 있는 워드에 대한 포인터를 정의하려면 포인터를 정의할 때 __packed 한정자를 사용하여 다음과 같이 지정해야 한다.

```
__attribute__((packed)) int *p; // pointer to unaligned int
```

그러면 EISC 컴파일러에서는 LD를 사용하지 않고 포인터의 정렬 방식에 관계없이 올바른 값을 액세스하는 코드를 생성한다. 이렇게 생성된 코드는 컴파일 옵션에 따라 연속된 바이트 액세스 코드일 수도 있고 변수 정렬 방식에 따른 시프트 및 마스킹 코드일 수도 있어 시스템 성능이 저하되고 코드 크기에 좋지 않은 영향을 주게 된다.

1.3.2 구조체의 정렬되지 않은 필드

전역 변수가 해당 기본 크기 경계에 있는 것과 같이 구조체의 필드도 해당 기본 크기 경계에 있다. 따라서 컴파일러에서 필드 사이에 패딩(padding)을 삽입하여 필드를 정렬해야 하는 경우가 있다.
EISC 컴파일러에서는 특정 구조체의 정렬 방식을 알고 있는 경우 액세스하려는 필드가 패킹된 구조체 내에 정렬되어 있는지 여부를 확인할 수 있다. 이러한 경우 컴파일러에서는 가능하면 보다 효율적으로 정렬된 워드 또는 하프워드(half-word) 액세스를 수행한다. 특정 구조체의 정렬 방식을 알 수 없으면 LD, ST 등 여러 개의 정렬된 메모리 액세스와 고정된 시프트 및 마스킹을 함께 사용하여 메모리의 올바른 바이트에 액세스한다.
정렬된 필드와 정렬되지 않은 필드를 알 수 있도록 컴파일러에 추가 정보를 제공할 수도 있다.

이렇게 하려면 정렬되지 않은 필드를 __packed로 선언하고 구조체 자체에서 __packed 특성을 제거해야 한다. 이 방법은 권장되는 방법일 뿐 아니라 구조체 내의 기본 방식으로 정렬된 구성원에 빠르게 액세스할 수 있는 유일한 방법이다.

예1-1)

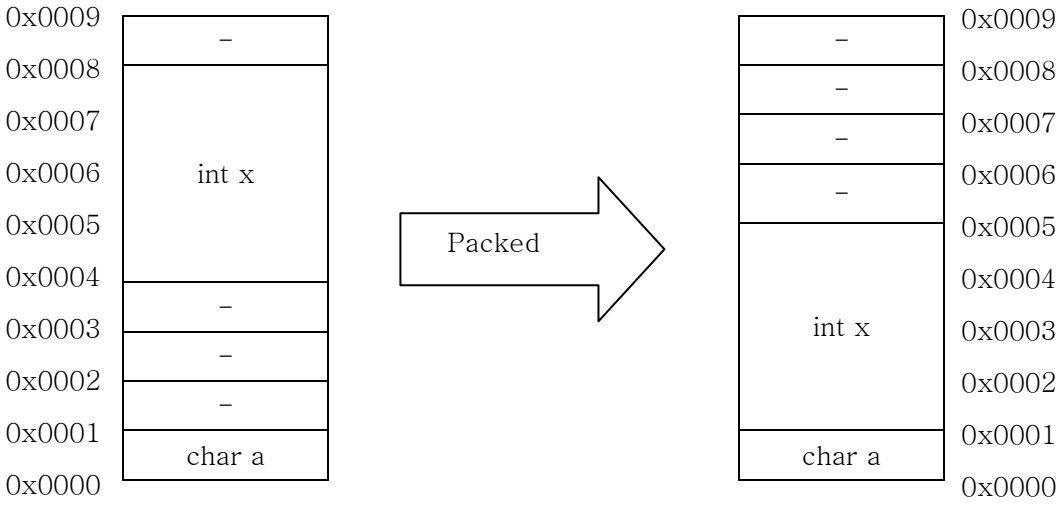
```
struct foo
{
        char a;
        int x[2];
};
```

예1-1)은 __packed를 사용하지 않고 구조체 foo를 사용할 경우 int형에 따라 4byte 단위로 정렬된다. 구조체 정렬과정에서 char형 변수는 4byte로 할당되고 이 과정에서 생기는 빈 공간은 의미 없는 값을 채우는 패딩을 통해 4byte단위로 정렬된다.
예1-1)에서와 같은 빈 공간의 낭비를 없애기 위한 방법으로 예1-2)와 같이 __packed를 통해 데이터를 정렬할 수 있다.

예1-2)

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

예1-2)의 경우 구조체 내의 정렬되지 않은 int형 변수는 __packed를 사용해 패딩을 통해 생기는 빈 공간의 낭비를 줄일 수 있다. 아래의 [그림 1-2]는 정렬된 구조체의 메모리 할당의 예와 예1-2)의 구조체 즉 packed 된 구조체의 메모리 할당 예시이다.



[그림 1-2] 예)1-1과 예)1-2의 구조체

공용체에도 이와 같은 방법을 사용할 수 있다. 즉, 메모리에서 정렬되지 않는 공용체의 구성 요소에 __packed 특성을 사용할 수 있다.

1.4 임베디드 소프트웨어 개발

1.4.1 C, C++ 및 어셈블리 언어 조합

개별적으로 컴파일되고 어셈블된 C, C++ 및 어셈블리 언어들은 링킹(linking)을 통하여 프로그램에 함께 사용할 수 있다. 또한 C/C++ 코드 내에 인라인 어셈블리(Inline assembly)로 작성 가능하다. 이러한 루틴은 EISC 컴파일러의 어셈블러를 사용하여 어셈블 된다. 그러나 어셈블러를 사용할 경우 작성할 수 있는 어셈블리 언어 코드에는 여러 가지 제한이 있으며 자세한 내용은 **4장 C,C++ 및 어셈블리 언어를 이용한 프로그래밍**을 참조하면 된다.

1.4.2 프로세서 Exception

EISC 프로세서에서는 다음과 같은 타입의 Exception이 인식된다.

EXCERTION	설 명
RESET	코어 프로세서의 초기화 동작을 요구하고자 할 때, 외부에서 코어 프로세서로 입력해 주는 신호를 통하여 발생한다.
External Hardware Interrupt	외부 하드웨어 모듈에서 요청하는 인터럽트를 의미한다. Autovectored Interrupt와 Vectored Interrupt 의 두 가지 경우가 있다.
Software Interrupt (SWI)	소프트웨어적으로 발생시키는 인터럽트를 의미한다. 관리자의 자원을 사용자 모드에서 접근하고자 하는 경우 호출한다.
Non-Maskable Interrupt	Non-Maskable Interrupt(NMI)는 인터럽트 컨트롤러에서 마스킹(인터럽트 요청을 거부하는 것) 동작을 할 수 없는 인터럽트를 의미한다.
System Co-Processor Interrupt	시스템 보조 프로세서 인터럽트는 메모리 접근 과정에서 발생하는 인터럽트를 의미한다.
Coprocessor Interrupt	보조 프로세서 인터럽트는 보조 프로세서의 접근 과정에서 발생하는 인터럽트들을 의미한다.
Breakpoint & Watchpoint Interrupt	OSI에서 제공하는 디버깅 기능을 제공하기 위한 인터럽트이다.
Bus Error & Double fault	명령어 버스나 데이터 버스의 복구 불가능한 버스의 오류인 경우에 Bus Error가 발생하며, 인터럽트 핸들러 Fetch 혹은 PUSH 과정에서 오류가 발생한 경우 해당 인터럽트를 더 이상 처리할 수 없으므로 Double fault가 발생한다.
Undefined Instruction Exception	정의되지 않은 명령이 입력된 경우 발생하는 예외이다. 이 경우 버전에 따라 오류를 발생시키거나, NOP로 처리 할 수 있다.

Unimplemented Instruction Exception	명령어 셋 상에서 정의되어 있으나, 해당 버전에서는 구현되어 있지 않은 명령을 의미한다.

[표 1-1] 인터럽트의 종류와 의미(AE32000의 경우)

EISC 프로세서는 대부분의 Exception을 지원하며 지원되는 Exception과 인터럽트에 대한 자세한 내용은 **5장 Exception Handling**을 참조하면 된다.

2 장

C/C++ 컴파일러 & Binutils

이 장에서는 C/C++ 컴파일러를 사용하는 방법과 Binutils에 대해 설명한다. 여기에는 다음 단원이 포함되어 있다.

- C/C++ 컴파일러
- Binutils

2.1 C/C++ 컴파일러

이 장에서는 EISC 컴파일러의 기본적인 사용법 및 옵션을 다루고 있다.
EISC 컴파일러는 GCC를 기반으로 하고 있으며 기본적인 사용법이나 옵션은 GCC에서 사용하는 것과 동일하다. 컴파일러 설정 정보를 보거나, 전체적인 동작을 제어하기 위해서 여러 가지 명령 행 옵션을 쓸 수 있다. 또한 -o 같은 한 글자 옵션과 -nostartfiles 같은 여러 글자 옵션을 모두 사용할 수 있다.
EISC 컴파일러를 사용하여 프로그램을 컴파일하거나 링크(link)하는 방법과 각 옵션에 대해서 간단히 설명한다. 프로그래밍 기법에 대한 자세한 내용은 **3장 스타트업, 4장 C,C++ 및 어셈블리 언어를 이용한 프로그래밍**을 참조하면 된다.

일반적으로 컴파일을 수행하기 위해 EISC 컴파일러를 호출하는 방식은 다음과 같다.

```
$ae32000-elf-gcc [Option] <Input_file>
```

이 장에서는 AE32000을 사용하여 컴파일하는 방법을 예로 들고 있다. 다른 프로세서를 사용하는 경우(SE2608, SE3208, AE64000)에는 (Target_name)을 사용하는 프로세서 이름으로 바꾸어 사용하면 된다.

2.1.1 Object 파일 만들기

컴파일은 소스의 각 파일들을 컴파일하여 목적 파일로 만든 다음 이 파일들을 결합해 실행 파일로 만드는데, 만일 특정 파일에 오류가 있다면 그 파일의 소스만 재 컴파일해 다른 파일과 결합해서 실행 파일을 만들면 된다. 여기서는 각 소스 파일들을 컴파일 하는 과정을 설명한다.

```
$ae32000-elf-gcc -c -o main.o main.c
```

이 명령에서 '-c' 옵션은 컴파일을 실행하면 main.o 라는 목적 파일을 생성할 수 있다. 이 옵션은 어셈블 과정까지만 수행하고 linking 과정은 수행하지 않는 옵션이다. '-o <output_filename>'옵션은 입력 파일에 대한 출력 파일의 이름을 지정하는 옵션이다. Object 파일을 만드는 과정에서 '-o' 옵션을 사용하지 않아도 입력 파일과 같은 이름의 Object 파일이 생성된다.

2.1.2 디렉토리 경로 지정하기

EISC 컴파일러에서 디렉토리 검색 경로를 조작하는 방법에 대해 설명한다. 프로그램 컴파일을 실행하는 과정에서 사용자는 헤더파일과 라이브러리 파일을 직접 작성해 사용하는 경우가 있다. 이럴 경우 옵션을 사용해 원하는 디렉토리 경로를 지정할 수 있다. AE32000 컴파일러를 설치했다면 기본적인 헤더파일은 /usr/local/ae32000-elf/include 디렉토리에 설치되어 있고 라이브러리 파일은 /usr/local/ae32000-elf/lib 디렉토리에 설치되어있다.
디렉토리 경로를 가리키는데 있어서 절대경로와 상대경로를 사용해 디렉토리를 지정할 수 있다.

절대경로는 최상위 디렉토리인 루트 디렉토리로부터 경로를 나타내는 것이고 상대경로는 현재 자신의 위치로부터 경로를 나타내는 것이다. 상대경로에서 ‘./’는 현재의 디렉토리, ‘../’는 바로 한 단계 위의 디렉토리를 가리킨다.
다음 명령은 main.c에 포함되어야 하는 헤더 파일과 라이브러리 파일의 절대경로와 상대경로에 대한 디렉토리 지정 방법에 대한 예이다.

헤더 파일과 라이브러리 파일의 상대경로가 ./project(현재 AE32000에 있는 경우) 일 경우

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

헤더 파일과 라이브러리 파일의 절대경로가 cygwin/ae32000/project 일 경우

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

위의 경우는 Cygwin설치 폴더내에 사용자의 헤더파일과 라이브러리파일이 있을 경우의 절대경로를 이용한 디렉토리 지정방법이다.

헤더 파일과 라이브러리 파일의 절대경로가 /cygdriv/d/ae32000/project 일 경우

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

위의 경우는 사용자의 헤더파일과 라이브러리파일이 Cygwin설치 폴더가 아닌 다른 폴더에 있는 경우의 절대경로를 이용한 디렉토리 지정방법이다.

이 명령에서 ‘-I’ 옵션은 헤더파일 검색 경로를 추가하며 표준 검색 목록 앞 부분에 추가된다. 옵션을 여러 번 사용해 다른 검색 경로도 추가할 수 있다. 여기서는 /project/header/ 에서 사용자가 작성한 헤더 파일을 포함해 컴파일 한다. ‘-L’ 옵션은 라이브러리 검색 경로를 추가하게 된다. 여기서는 /project/library/ 에서 사용자가 작성한 라이브러리 파일을 포함해 컴파일 하게 된다.

2.1.3 Linking

Linking은 컴파일 과정에서 맨 마지막 과정으로 여러 개의 Object 파일을 하나의 실행 가능한 형식의 파일로 합치는 일을 한다. 이 과정에서 표준 시스템 스타트업 파일과 라이브러리, 사용자가 작성한 스타트업 파일과 라이브러리를 사용해 실행파일을 만들게 된다.

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

이 명령에서 crt0.o 파일은 EISC 프로세서를 타겟으로 작성된 스타트업 코드(crt0.S)의 Object 파일이다. ae32000.vct 파일은 Linker Script File로서 프로그램 코드와 데이타의 배치 등을 결정해 준다. 또한 각 섹션의 위치와 크기도 지정해 줄 수 있다. ‘-g’ 옵션은 소스 레벨 디버거인 GDB를 사용하기 위해 디버깅 정보를 생성하라는 옵션이며 ‘-Wall’ 옵션은 모든 경고 메시지를 출력하도록 한다. 보통은 ‘-g’, ‘-Wall’ 옵션을 주고 컴파일 하는 것이 디버깅에 도움을 주기 때문에 항상 사용하는 것을 권장한다. Object 파일로 컴파일 할 때와 달리 Linking 과정에서의

‘-o’ 옵션은 생성할 실행 파일의 이름을 정해준다. 여기서는 ‘main.elf’라는 파일을 생성한다. 만일 ‘-o’ 옵션을 사용하지 않는다면 ae32000-elf-gcc는 ‘a.out’라는 실행 파일을 만들게 된다. Linker 단계에서 추가적인 옵션으로 표준 Startup 파일과 라이브러리를 사용하지 않으려면 -nostdlib(-nostartfile 과 -nodefaultlib을 지정한 것과 동일), -nostartfiles, -nodefaultlib 의 옵션을 사용하면 된다.

최적화옵션은 코드의 목적과 최종 결과는 바꾸지 않으면서 효율적으로 바꾸는 것이 목적이다. EISC 컴파일러에서도 일반적인 최적화 옵션으로 ‘-O2’ 사용하는데 ‘-O2’ 는 코드의 크기와 실행 시간을 절충한 최적화 코드를 만들 수 있다.

EISC 컴파일러에 사용되는 옵션은 GCC의 옵션과 동일하므로 자세한 옵션 사용법과 여러 가지 옵션에 대한 정보를 얻고자 한다면 GCC 매뉴얼을 참고하면 된다.

2.2 Binutils

각종 Object 파일 포맷 들을 조작하기 위한 프로그래밍 도구 모음으로 여러 개의 목적 파일을 하나로 연결 하고, 라이브러리 관리를 위한 링커와 어셈블러를 포함한다.
GNU Binary utilities 는 ELF Object Code Format을 지원한다. ELF Format은 32-bit CPU를 위한 Object code로 널리 사용되고 있으며 Binutils에 관한 자세한 내용은 GNU manual [Binutils]을 참고하면 된다.

다음은 Binutils에 포함된 도구 모음을 설명한다.

도구 모음	설 명
ae32000-elf-addr2line	프로그램 주소를 파일명과 줄 변호로 변환한다.
ae32000-elf-ar	아카이브를 만들고 수정한다.
ae32000-elf-as	어셈블러이다.
ae32000-elf-ld	실행 파일을 만들어 주는 링커이다.
ae32000-elf-nm	Object 파일 objfile...의 심볼을 출력한다.
ae32000-elf-objcopy	Object 파일 내용을 다른 Object 파일로 복사하거나 입력 Object 파일과 다른 형식으로 출력 파일을 작성할 수 있다.
ae32000-elf-objdump	Object 파일에 대한 정보를 출력한다.
ae32000-elf-ranlib	아카이브의 색인을 만들어서 아카이브에 저장한다.
ae32000-elf-readelf	ELF 형식 Object 파일에 대한 정보를 출력한다.
ae32000-elf-size	Object 파일이거나 아키이브 파일인 objfile의 각 섹션의 크기와 총합을 출력한다.
ae32000-elf-strings	주어진 각 file에서 출력 가능한 문자가 최소 4개 (혹은 아래 옵션에서 지정한 수만큼) 연속되어 있으면 출력한다.
ae32000-elf-strip	Object 파일 objfile에서 모든 심볼을 제거한다.

[표 2-1] Binutils 도구 모음

3 장
스타트업

이 장에서는 메모리 맵과 초기화 코드 등에 관한 설명을 한다.
초기화 코드는 애플리케이션이 실행되기 전에 소프트웨어가 동작할 수 있는 설정을 해주는 것이다. 마치 자동차를 운전하기 위해 시동을 거는 것과 비슷한 역할이다.
여기서는 다음 단원이 포함되어 있다.

- 프로그램 메모리 맵
- crt0.S
- crt1.c
- Linker Script
- “Hello World!” 만들기

3.1 프로그램 메모리 맵

3.1.1 text 영역

“.text”영역은 CPU를 동작시키는 명령어가 들어가 있는 영역이다. 컴파일러가 만들어낸 C 소스로부터 번역한 기계어 명령어들이 해당위치에 속해있으며 상수로 선언된 변수나 읽기전용의 데이터도 이 영역에 포함된다 읽기전용의 데이터는 명령어와 동일하게 취급되어 ROM과 같은 쓰기가 불가능한 물리 메모리에 저장되어 있어도 프로그램 동작 시 아무런 문제가 발생하지 않는다.

3.1.2 data 영역

“.data”영역은 초기화된 전역변수와 배열 및 static으로 할당된 지역변수가 들어가는 영역이다. 컴파일러는 C소스로부터 선언과 동시에 초기값이 지정된 전역변수들과 선언과 동시에 초기값이 지정된 static 선언된 지역변수들을 바이너리 파일의 한 부분에 위치시킨다. 이 부분은 스타트업 코드가 .data 영역으로 복사한다.

3.1.3 bss 영역

“.bss”영역은 초기화되지 않은 전역변수가 할당되는 영역으로 컴파일러가 생성한 바이너리 파일에 특정한 영역을 차지하고 있지는 않으나 프로그램이 실행될 때 메모리에선 특정한 영역을 차지하게 된다. 이 영역은 굳이 ‘0’등의 값으로 초기화를 하지 않아도 큰 문제는 없지만 프로그래머들이 흔히 사용하는 실수인 초기화 하지 않고 선언한 변수의 값은 ‘0’으로 가정하는 실수를 예방하기 위해서 ‘0’으로 초기화 해주는 것이 좋다.

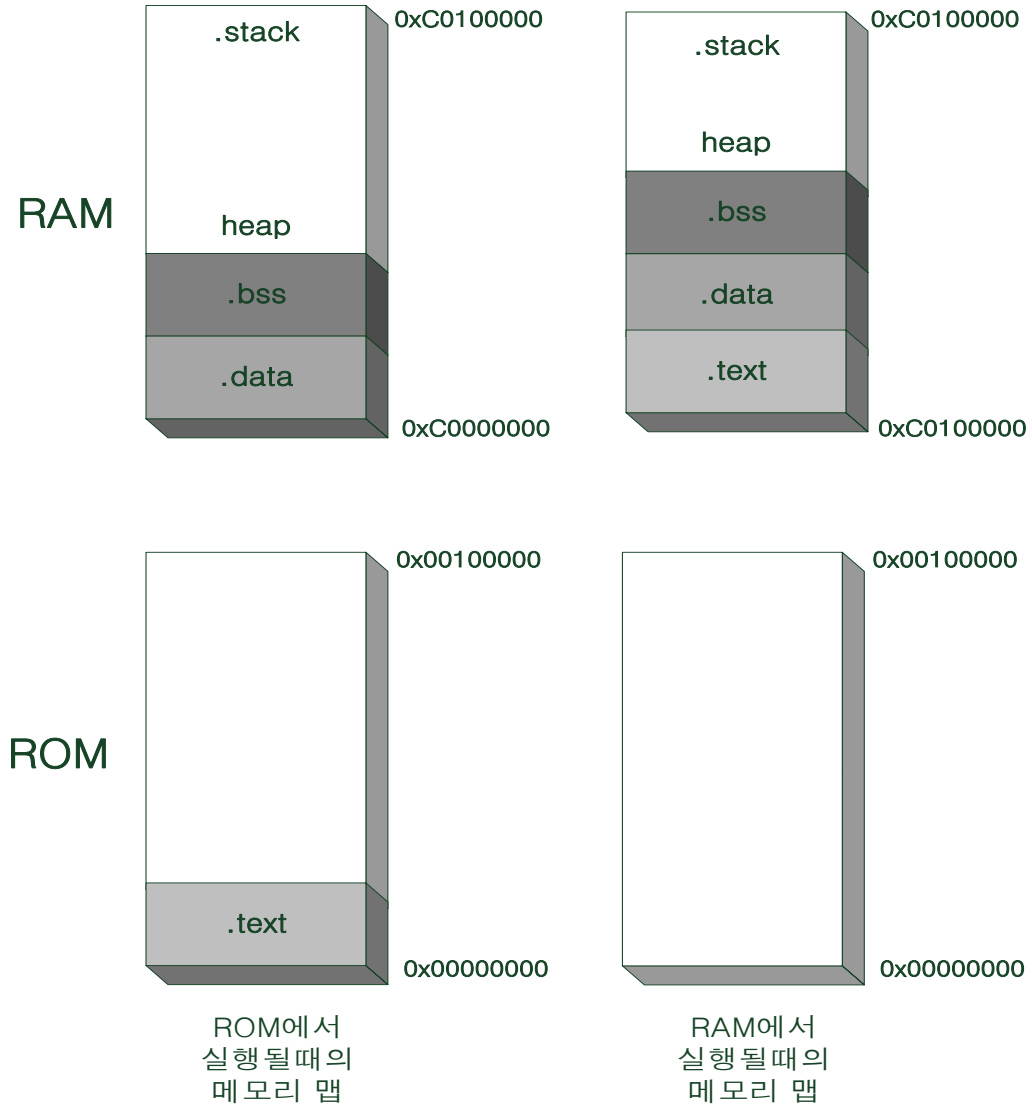
3.1.4 stack 영역

“.stack”영역은 EISC 프로세서의 스택 사용이 다운워드(downward) 타입이기 때문에 메모리의 끝부분으로 지정해주게 된다. 스택은 지역변수, 인자 등이 임시로 들어가는 영역으로 프로그램 동작 시 매우 빈번하게 사용되는 영역이다. 스택의 크기는 함수호출의 양과 예외처리 등에 의해 수시로 변하게 되므로 특정한 크기를 결정하지는 못하고 스택의 시작점만을 지정하여 FILO(First In Last Out)의 형태로 사용하게 된다. 사용하는 스택의 양이 커지고 아래에 설명할 heap 메모리 양이 커져서 두 메모리가 서로 겹쳐지게 되면 메모리 충돌이 발생하여 프로그램 오류가 발생하게 된다.

3.1.5 heap 영역

미리 결정된 변수나 배열이 아니라 프로그램 중간에 필요한 만큼의 메모리를 할당하여 사용할 수 있는 기능을 위해서 예비된 영역이다. C프로그램에서 malloc/free 함수를 이용하여 할당받는 메모리 영역이 heap 메모리 영역이다. 업워드 타입으로 사용되며 heap메모리의 사용이 많아지게 되어 .stack 영역을 침범하게 되면 프로그램은 오작동하게 된다.

3.1.6 메모리 맵



[그림3-1] - 프로그램 실행시 메모리맵

ROM에서 실행되는 코드의 메모리 맵의 경우 위의 좌측 그림에서처럼 읽기만 필요한 .text 영역은 ROM에 위치해 있고 읽고 쓰기가 필요한 변수 및 스택 영역은 RAM에 위치해 있게 된다.

RAM에서 실행되는 코드의 경우에는 실행코드인 .text 영역 조차도 RAM에 위치해 있게 된다.

3.2 Linker Script

3.2.1 Linker Script

Linker script는 output format, output architecture, start entry, include library, library 및 header file search directory, memory map, section name의 정보를 포함한다. linker는 이와 같은 linker script의 정보를 이용하여, 링커의 input인 목적파일들의 section을 재배치한다. 각 section은 linker script에서 정의된 메모리 맵에 할당되며, provide로 명시된 심볼들은 심볼테이블에 그에 해당하는 VMA와 함께 저장된다. 또한, SEARCH_DIR의 경로를 참조하여 필요한 라이브러리를 함께 링킹하며, 그 라이브러리 역시 section정보에 의하여 재배치된다.

3.2.2 Linker Script의 예

ROM에서 프로그램이 실행되는 경우, 아래의 [표3-1]과 같이 ROM/RAM의 크기를 지정하여 text section은 ROM에 위치하고, data/bss section은 RAM에 위치하도록 작성하면 된다.
Linker script file은 '-T' Option을 통하여 linker가 참조하고 section을 재 배치한다.

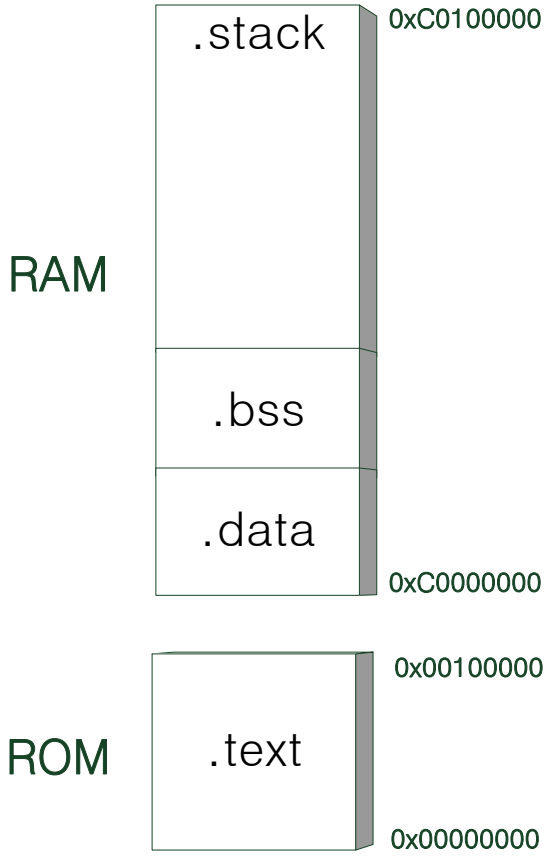
```
OUTPUT_FORMAT("elf32-ae32000-little", "elf32-ae32000-little",
                              "elf32-ae32000-little")

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));
```

[표 3-1]

위의 링커 스크립트 파일대로 메모리를 구성했을 때는 아래와 같다.



[그림3-2] 표 3-1의 링커 스크립트가 구성한 메모리 맵

이 File에는 메모리의 크기와 위치를 지정하는 부분과 프로그램과 변수의 초기화 값이 저장된 부분, Debug정보를 저장하는 부분 등을 정의하는 부분이 있다. 이 중 run time시 메모리의 위치와 크기를 지정하는 부분은 MEMORY로 정의되어 있다.
상세하게 예로서 AE32000의 Loader Script의 각 항목을 분석하여 보겠다.

첫 줄에 OUTPUT_FORMAT("elf32-ae32000", "elf32-ae32000", "elf32-ae32000")라는 항목이 있다. OUTPUT_FORMAT Command는 출력 File에 어떤 BFD Format이 사용될 것인가를

결정한다. 세 인수가 있는데 두 번째 인수는 명령 행(Command Line) 인수에서 -EB로 지정되었을 때, 즉 Big Endianness로 정해졌을 때 사용될 BFD Format 을 지정한다. 세 번째 인수는 -EL로 Little Enidanness로 지정되었을 때 사용될 Format 을 지정한다.
첫 번째 인수는 -EL, -EB의 어떤 Option도 주어지지 않았을 때 사용될 Default BFD Option이다. 여기서는 어떤 Option에도 상관없이 elf32-ae32000 BFD Format을 사용한다. 다음 줄의 OUTPUT_ARCH(ae32000)은 Target Machine의 Architecture를 정의한다. 인수는 BFD Library가 사용하는 이름들 중의 하나가 되면 이 경우는 AE32000이 된다. (TargetName)-elf-objdump에 -f Option을 주어 정의된 Target Machine의 Architecture를 볼 수 있다.

SEARCH_DIR(/usr/local/ae32000-elf/lib) 명령어는 Loader가 Library를 찾기 위해 탐색하는 Directory를 지정한다. 이것은 명령 행에서 -L Option으로 지정하는 것과 같은 효과이다.
MEMORY 명령어는 Target System에 RAM과 ROM의 두 종류의 메모리가 있으며 각각의 시작 주소와 메모리 크기를 정의한다.

SECTIONS 명령어는 출력 File의 Memory Layout을 정의한다. Section은 Loader가 프로그램의 Block 을 run-time address로 옮길 때 이용되며 이 때 Section의 길이나 순서는 바뀌지 않는다. 가장 간단한 Loader Script는 SECTIONS 명령어만 가지고 있을 수 있다. 위의 예에서는 출력 File의 .text Section은 모든 입력 File들(*은 모든 입력 File 을 의미)
의 .vects, .text, .stub 등의 Section들을 포함하게 된다. 그리고 .=0x0 이란 명령어가 .text앞부분에 위치하는데 이는 Location Counter가 0으로 Setting되는 것을 의미하며 즉 출력 File의 .text Section의 시작 주소는 0 번지가 되는 것이다. .text Section다음으로 .data, .bss Section들이 오는데 특별히 지정해 주지 않았기 때문에 이들의 시작 주소는 먼저 지정된 Section의 바로 다음 번지가 된다.

___ctors, ___ctors_end, ___dtors, ___dtors_end는 c++의 global constructor와 destructor에 대한 포인터들이 저장될 주소를 지정한다. 이러한 global constructor들은 __main(즉 main() 함수) 함수가 호출될 때 호출되게 되고 global destructor들은 exit시 또는 atexit() 함수 호출 시 호출된다.

. text Section 의 마지막 부분에 >rom 이라는 Command가 있는데 .dat 와 .bss의 >ram 과 대비된다. 이는 .text Section은 ROM에 .data와 .bss는 RAM에 위치해야 한다는 것을 정의한다.

Loader Script의 여러 곳에 Section의 시작 위치와 끝나는 위치를 저장해 두는 _end나 _bss_start 등의 Symbol을 볼 수 있는데 이러한 값들을 저장할 시 사용자의 Code에서 같은 이름의 Symbol을 사용할 수 없게 된다. 이것을 허용하고 싶을 때는 Provide Command를 쓰게 된다. 즉 마지막 줄에 PROVIDE(___data_size = SIZEOF(.data))라는 명령어의 경우 Loader Script에서 Data Section의 크기를 저장하기 위해 ___data_size라는 Symbol을 정의하였는데 만약 사용자 프로그램에서 같은 이름의 Symbol을 만들어 정의하여 사용하는 것이 허용되게 된다.

3.3 crt0.S

3.3.1 개요

CRT0는 C Runtime Library 0라는 뜻으로 실제 C Code중에서 제일 먼저 실행되는 Code라는 의미를 가지며 Assembly언어로 쓰여진 crt0.S라는 File로 저장되어 있다. 이 Library의 역할은 프로그램이 메인 함수로 들어가기 전에 스택포인터 및 시스템 컨트롤에 필요한 모든 초기화 작업을 수행하는데 필요하다. 물론 대부분의 작업은 main()함수 내에서 수행할 수 있기 때문에 필요한 최소한의 기능만을 crt0.S에 할당하고 나머지 작업을 main()함수에서 처리할 수 있다.

crt0.S의 주요기능은 다음과 같다.
Stack pointer register 초기화
data 섹션 복사
bss 섹션 초기화
C++사용시 Constructor 초기화
Main() 호출
Exit() 호출

3.3.2 소스코드 및 설명

RESET 상태를 벗어나면 프로그램은 Reset Vector가 가리키는 crt0.S로 들어간다.
제일 먼저 하는 것은 Stack Pointer register의 초기화 작업이다. 이때 사용되는 Stack Pointer의 값은 Symbol(_stack)로 처리되어 있으며 그 실제 값은 linker script에 들어있다.

```
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));
```

[표 3-2]

Stack Address 0xc00ffff0는 사용하고자 하는 메모리의 마지막 주소를 사용한다. 마지막 주소를 사용하는 이유는 EISC CPU는 Downward 스택을 사용하기 때문이다. 따라서 _stack이라는 값으로 linker script에서 실제 값이 명시 되어 있으며 그 값이 crt0.S File에서 사용된다.

```
_START:
/* initialize stack pointer : move unline from GPR upline*/
        ldi _stack-8, %r8
        mov %r8,%sp
……
```

[표 3-3]

%R8 값을 Stack Pointer에 저장 함으로서 Stack Pointer의 초기화 작업이 종료된다.
EISC compiler는 Downward Stack을 기본으로 사용하므로 사용하고자 하는 RAM Address의 최상위 Address부터 할당한다.

```
        jal ___main      # data, bss 섹션 초기화 함수 호출
        jal _main        # main함수 호출
```

[표 3-4]

main함수 이전에 crt0.S에서 호출하는 위의 함수들은 crt0.S에서 어셈블리어로 작성할 수도 있고 아래의 예처럼 C 함수로 작성할 수도 있다.

3.4 crt1.c

3.4.1 개요

crt1.c는 crt0.s에서 호출하는 초기화 함수를 모은 소스파일이다.

3.4.2 소스코드 및 설명

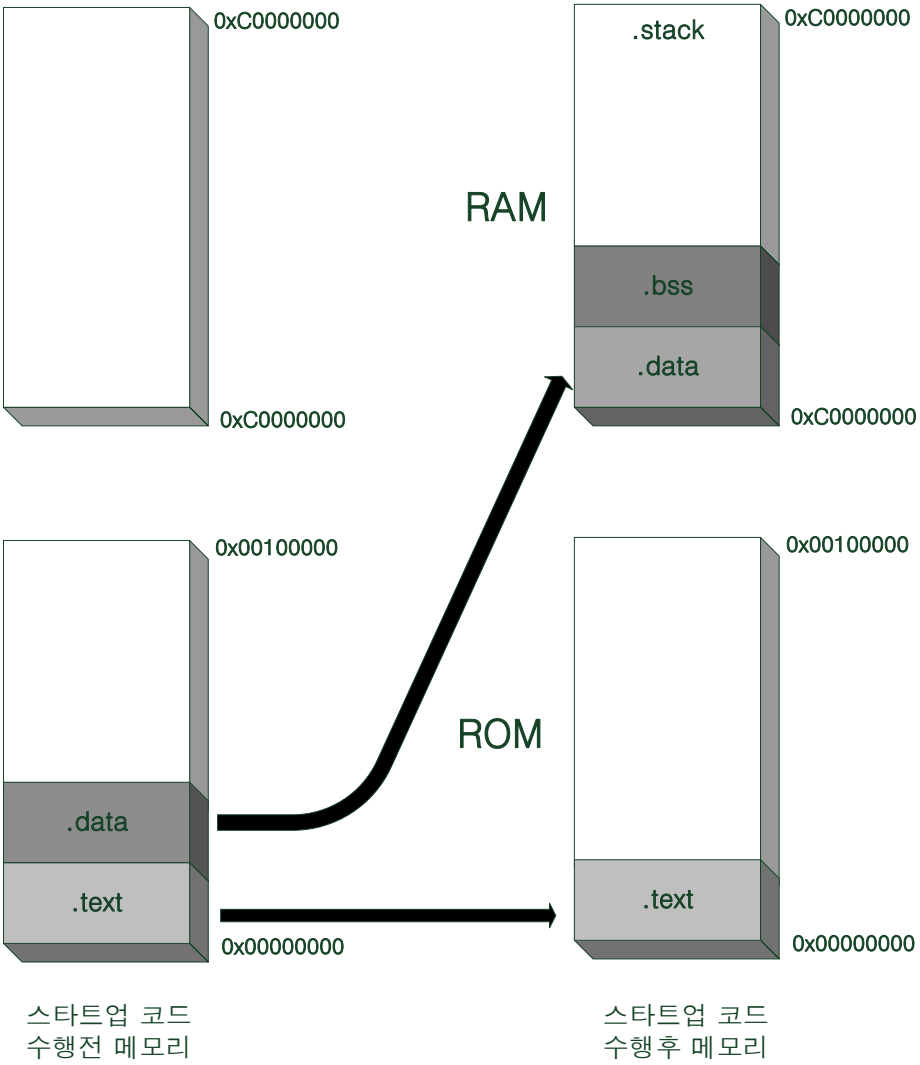
```
extern unsigned char __shadow_data[];
extern unsigned char __data_reload[];
extern const      int   __data_size[];
extern const      char _bss_start;
extern const      int __bss_size[];

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;                  // ROM의 data영역 시작주소
  destptr = (char *)&__data_reload;                 // RAM의 data영역 시작주소
  count     = (int)&__data_size;                    // data영역 길이
  memcpy(destptr, srcptr, count);                   // data영역 값 복사


  /* bss section initialization with zero value */
  srcptr = (char *)&_bss_start;                     // RAM의 bss영역 시작주소
  count = (int)&__bss_size;                         // bss영역 길이
  while(count--)*srcptr++ = 0;                      // bss영역 초기화
```

```
  for (p = __ctors_end; p > __ctors; )          // main함수 이전 constructor 속성의
  (*--p) ();                                    // 함수 실행
}
```

[표 3-5]

위의 소스코드에서 본 것과 같이 crt0.S에서 호출한 __main()함수는 bss영역 초기화 data영역 초기화를 수행한다.

crt0.S와 crt1.c가 수행되기 전과 후의 메모리 사용은 아래와 같다.



[그림3-3] 스타트업 코드 실행 전/후 비교

위에서 보듯이 스타트업 코드가 수행되기 전의 ROM에 바이너리 형태로 저장되어 있는 텍스트 코드는 제 위치에 그대로 있으며, ROM에 바이너리 형태로 저장되어 있는 초기화된 전역변수 영역인 .data는 읽기/쓰기가 가능한 RAM으로 복사된다. 또, 초기화 되지 않은 전역변수 영역인 bss영역은 해당 크기만큼 0으로 초기화하게 되며 이후의 영역은 프로그램에서 .stack영역과 .heap메모리 영역으로 사용하게 된다.

3.5 “Hello World!” 만들기

3.5.1 “Hello World!”

지금까지 언급한 내용을 토대로 “Hello World!”가 출력되는 프로그램을 만들어보자.
파일은 crt0.S crt1.c main.c 3개의 파일로 구성되어 있다.

3.5.2 Hello World를 위한 ae32000.vct

위에서 설명한 3.4.2의 소스코드와 같다.

3.5.3 Hello World를 위한 crt0.S

```
##-------------------------------------------------------------------
        .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
```

[표 3-6]

3.2.2의 소스코드와 같으며 3.2.2에서는 생략되거나 코드를 나눠 놓았기 때문에 이장에서 하나로 합쳤다.

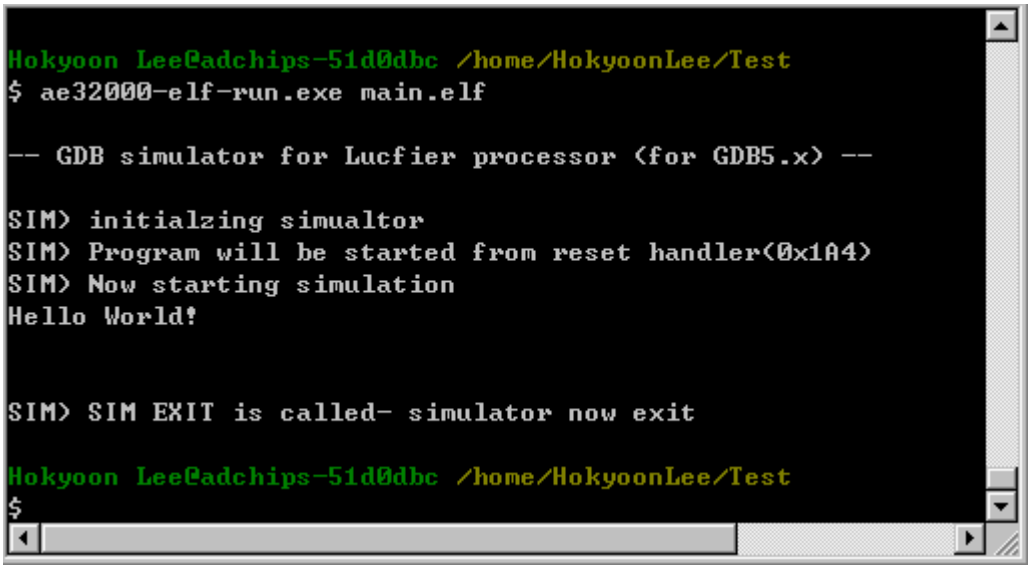
3.5.4 Hello World를 위한 main.c

```
#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);
}
```

[표 3-7]

3.5.5 Hello World Compile & Linking

위와 같이 파일을 작성한 후 컴파일 하고 실행하여 본다.
Cygwin상에서 GDB simulator를 사용한 실행결과는 아래와 같다.



4장
C,C++ 및 어셈블리 언어를 이용한 프로그래밍

이 장에서는 C,C++및 어셈블리 언어가 조합된 코드를 작성하는 방법을 설명한다. C,C++에서 인라인 및 임베디드 어셈블리를 사용하는 방법에 대해서도 설명한다. 여기에는 다음 단원이 포함되어 있다.

- 인라인 어셈블리 사용
- 어셈블리에서 C 언어 변수의 사용
- C/C++ 및 어셈블리 언어 간의 호출
- 표준 함수 호출

4.1 인라인 어셈블리 사용

EISC 컴파일러에 기본적으로 제공된 인라인 및 임베디드 어셈블리를 통해 C 또는 C++에서 직접 액세스할 수 없는 타겟 프로세서의 기능을 사용할 수 있다.

4.1.1 인라인 어셈블리의 기능

인라인 어셈블리는 C 및 C++과 매우 융통성 있는 인터워킹을 지원한다. 레지스터 피연산자는 임의의 C 또는 C++식일 수 있다. 또한 인라인 어셈블리는 복잡한 명령어를 확장하고 어셈블리 언어 코드를 최적화 한다.

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

[표 4-1] 인라인 어셈블리 명령 사용 예 (AE32000)

다중 최적화 컴파일러 옵션인 -01, -02 또는 ,-03중 하나를 사용할 경우 인라인 어셈블리 언어는 컴파일러에서 수행하는 최적화의 영향을 받는다.

4.2 어셈블리에서 C 언어 변수의 사용

C언어로 프로그램을 작성하다가 필요에 의해 인라인 어셈블리 언어로 프로그램을 작성할 때 C언어로 선언되거나 정의된 변수를 인라인 어셈블리 명령어 내부에서 직접 변수로 사용하려면 다음과 같은 문법을 사용한다.

4.2.1 C언어 변수의 사용 방법

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

[표 4-2] C언어 변수의 사용 포맷

[표 4-2]의 명령어 형식을 살펴보면 우선 시작은 일반적인 인라인 어셈블리 명령어와 같이 asm으로 시작해야 한다. Assembler template는 opcode와 operand의 string으로 구성되며, operand는 %0, %1 … 과 같이 표시된다. %0, %1 … 은 output/input operand의 표기된 순서를 의미한다. 첫 번째로 주어지는 변수 값을 지시하기 위해서는 '%0'이 사용되고, 두 번째로

주어지는 변수를 인자로 사용하고 싶다면 '%1'을 사용한다. 예를 들어 assembler template가 "add %0, %1" 이고, output operand가 "r"(tmp1), input operand가 "r"(tmp2) 이면, '%0'은 tmp1에 타겟된 레지스터를 의미하고, '%1'은 tmp2에 할당된 레지스터를 의미한다.
콜론은 그룹을 구분을 의미한다. Assembler template과 operand도 ':'로 구분됨을 알 수 있다. Output/input operand들은 ','로 구분된다.
위에서 output/input operand로 표현된 것은 변수의 종류와 변수 이름으로 구성된다. 변수의 종류는 compiler에서 정의된 register class를 기반으로 표기하면 되며, 레지스터(r), 글로벌 변수(g), 정수 상수(i), 메모리(m) 등이 있다. 만일 그 변수가 출력으로 사용된다면 속성을 나타내는 문자 앞에 '='이 덧붙는다. 즉, "=g" 과 같이 표현된다. (변수의 이름)은 C언어로 작성된 함수나 모듈 내에서 선언 또는 정의된 변수의 이름을 적는 부분이다.
Clobbered register list는 assembler template이 사용하게 되는 하드웨어 레지스터를 표시한다. 예를 들어 asm("mov %0, %**%R0**" : : "r"(a) : "**R0**")와 같은 inline assembly code를 보면 레지스터 R0는 destination이 되면서 compiler가 의도하지 않은 programmer 임의의 clobber가 된다. 이와 같은 경우에 clobber list에 register의 이름을 적어준다.

4.2.2 예제를 통하여 인라인 어셈블리에서 C 변수 사용법 익히기

```
#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 %0, %1" :"=g"(d): "r"(c));
        asm("st %%r1, %r0" : "=g"(d):);

        printf("₩n %x", d);
}
```

[표 4-3] 인라인 어셈블리 명령어에서 C 변수의 사용 예 (AE32000)

[표 4-3]은 C언어에서 선언한 변수를 인라인 어셈블리 명령어 안에서 사용한 예를 보여주고 있다. 위에서 %%r1이라는 표현에서 %가 두 개가 붙은 것은 EISC에서 레지스터의 표현인 %와 위에 설명된 문법 상의 % 두 개가 만났기 때문이다.

4.3 C/C++ 및 어셈블리 언어 간의 호출

C++에서 C 및 어셈블리 언어 코드를 호출하고 C 및 어셈블리 언어에서 C++ 코드를 호출하는 방법을 설명한다.

4.3.1 C 소스 코드에서 어셈블리 함수 접근

이 절에서는 C 소스 코드의 내부에서 어셈블리 명령어들로 구성된 함수를 호출해서 사용하는 방법을 알 수 있다.
C 프로그램 소스 코드에서 다른 파일에서 선언되고 정의된 어셈블리 함수를 호출하기 위해서는 [표 4-4]에서와 같이 어셈블리 소스 코드내의 어셈블리 함수의 심볼을 '_' 문자로 시작해야 하며 Global로 선언해야 한다.

✓ 어셈블리 소스

```
# 0에서 10까지 더하는 함수
            .global _add        #add를 함수로 정의하는 부분
_add:                           #라벨
            push %r0, %r1
            ldi 0, %r0
            ldi 10, %r1
            ldi 0, %r8
add_loop:                       #반복될 부분의 라벨
            add 1, %r0          #1씩 더함
            add %r0, %r8
            cmp %r0, %r1
            jnz add_loop        #두 수의 비교가 0일 될 때까지 반복
            pop %r0, %r1
            jplr
```

[표 4-4] C 언어에서 어셈블리 코드로 된 함수 호출

✓ C 소스

```
#include <stdio.h>
__main_init(){};
extern int add();               //외부에서 가져다 쓸 함수 선언
int main(void)
{
            int c:
            c = add();          //함수 호출해서 C 변수에 저장
            printf("₩n %d",c);
}
```

[표 4-5] C 언어에서 어셈블리 코드로 된 함수 호출

위에서 [표 4-4]를 살펴보면 _add라는 심볼로 정의되고 바로 다음 줄에서 글로벌로 선언된 후에 [표 4-5]의 C 소스코드 안에서 함수로 호출되어 사용되는 것을 알 수 있다. 이때 두 File은 서로 연관성이 있게 되며 당연히 같이 Link되어야 할 것이다. 만일 어셈블리로 된 함수가 C 언어의 함수에서 변수 값을 전달 받거나 C언어의 함수에 연산 값을 되돌려주기 위해서는 C언어의 함수간 인자전달 방법을 따라야 한다.

```
c0700000 <_add>:

        .global _add                    #add를 함수로 정의하는 부분
_add:                                   # 라벨
        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:                       #반복될 부분의 라벨
        add 1, %r0              #1씩 더함
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            #두 수의 비교가 0일 될 때까지 반복
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();               //외부에서 가져다 쓸 함수 선언
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();              //함수 호출해서 C 변수에 저장
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
```

[표 4-6] [표 4-4, 4-5] 의 소스 코드를 Disassemble 한 결과(AE32000)

[표 4-6]에서 C Program Source Code에서 다른 File의 어셈블리 함수를 호출한 것이 맞는가를 확인할 수 있다.

4.3.2 언어 간 호출을 위한 일반적인 규칙

C 호출 규칙을 사용한다. C++에서 비 구성원 함수를 extern “c”로 선언하여 C 링키지를 포함하도록 지정할 수 있다. C 링키지를 포함한다는 것은 함수를 정의하는 심볼의 이름이 변환되지 않는다는 것을 의미한다. C 링키지를 사용하면 함수를 한 언어로 구현한 후 다른 언어에서 호출할 수 있다. Extern “c”로 선언된 함수는 오버로드 할 수 없다.

4.3.3 언어 간 호출 예제

① C++에서 C호출

[표 4-7]과 [표 4-8]은 C++에서 C를 호출하는 방법을 보여 준다.

```
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;
}
```

[표 4-7] C++에서 C 함수 호출

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

[표 4-8] C 에서 함수 정의

② C에서 C++ 호출

[표 4-11]과 [표 4-12]는 C에서 C++를 호출하는 방법을 보여 준다.

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

[표 4-11] C++에서 호출되도록 함수 정의

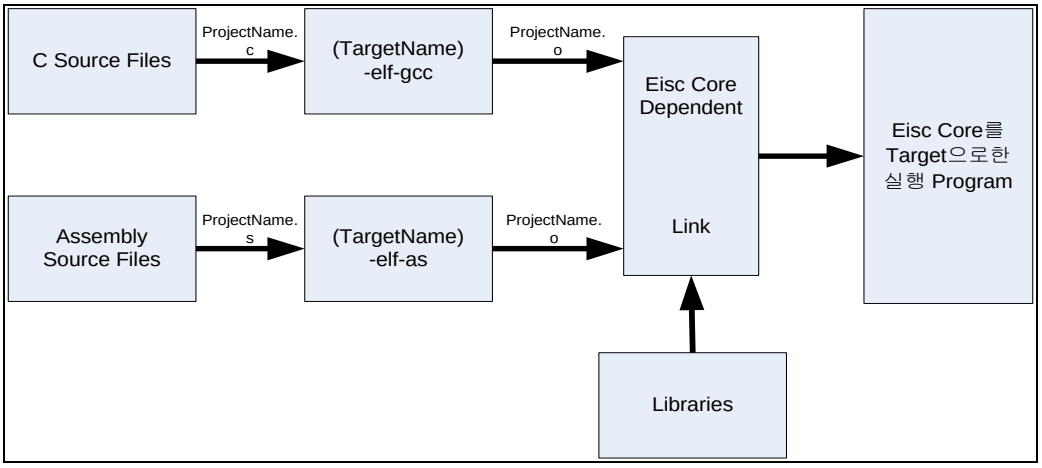
```
extern void cppfunc(int *i);
/* Declaration of the C++ function to be called from C */
int f(void) {
        int i=0;
        cppfunc(&i);                    /* call 'cppfunc' so it */
        return i * 3;
}
```

[표 4-12] C에서 함수 선언 및 호출

4.4 표준 함수 호출

4.4.1 표준함수 호출에 대하여

예를 들면 프로그램 수행시간을 줄여 코드의 수행결과를 좋게 하길 원할 때처럼 때때로 C또는 C++, 그리고 어셈블리언어의 혼합 프로그래밍이 필요할 때가 있다.
일반적으로 EISC 프로세서를 타겟으로 만든 어셈블리와 C그리고 C++의 소스 코드는 Object 파일로 변환이 되며 이 Object 파일들은 Link를 거쳐 EISC 프로세서에 의존한 실행 프로그램으로 생성이 된다([그림 4-1] 참조).



[그림 4-1] EISC 프로세서에 의존한 실행 프로그램의 생성

이 장에서는 이들 사이에 함수 또는 다른 모듈의 호출이나 결과 값의 Passing 등 표준 Procedure 호출에 대하여 알아보고 이 때 사용되는 레지스터 값들에 대해서 알아보도록 하자.

4.4.2 레지스터의 종류와 용도

EISC 프로세서는 종류별로 8개에서 16개의 일반 레지스터를 가지며 또한 종류별로 4개에서 7개까지의 특별한 레지스터를 가진다. EISC 프로세서는 모두 동일하게 16비트의 명령어 집합을 가지나 종류별로 레지스터의 크기는 16 – 64비트의 크기를 가진다.

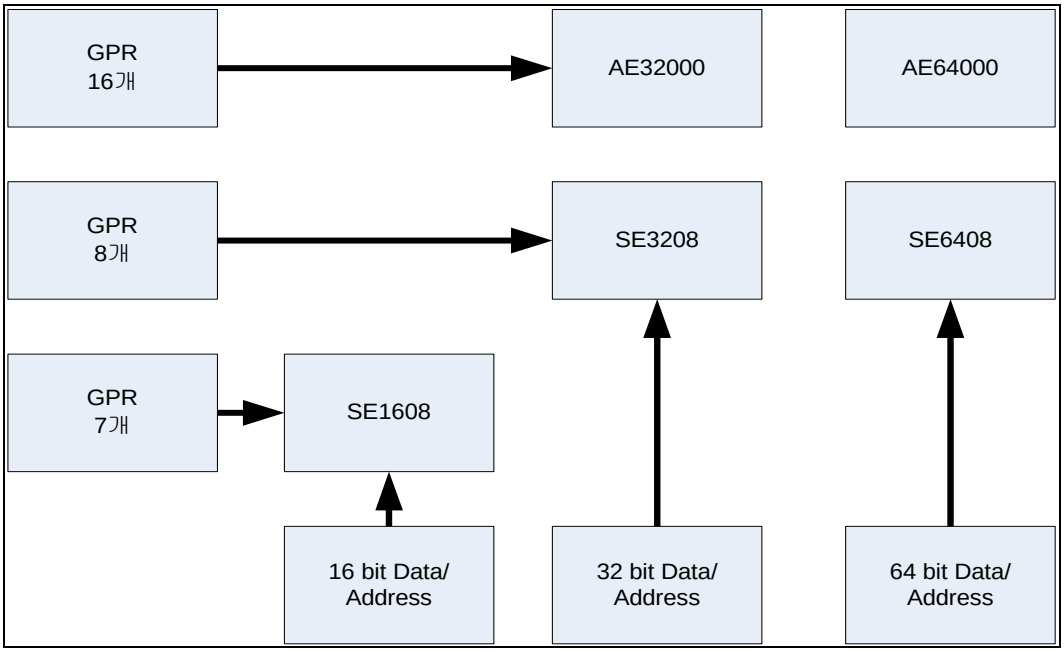
Register	사용 Core 예	내 용
R0 – R7	SE1608	%r0 ~ %r6 : GPR, %r7 : Stack Pointer(SP)
	SE3208	General Purpose Register
R8 – R15	AE32000이상 계열	General Purpose Register
PC	모든 EISC 프로세서	Program Counter
SP	모든 EISC 프로세서	Stack Pointer
USP	SE3208이상 계열	User Stack Pointer
SSP	SE3208이상 계열	Supervisor Stack Pointer
ISP	SE3208이상 계열	OSI Stack Pointer

LR	AE32000이상 계열	Link Register
ER	모든 EISC 프로세서	Extension Register
SR	모든 EISC 프로세서	Status Register

[표 4-15] EISC 레지스터

Bit	Core	내 용
63 – 32	AE64000 이상 계열	Reserved, always 0
31 – 16	AE32000 이상 계열	Reserved, always 0
15	SE3208 이하 계열	Reserved, always 0
	AE32000 이상 계열	Bit[15,9] : Processor Mode 선택 00 : Supervisor Mode 01 : OSI Mode 10 : User Mode
14	모든 EISC	NMI의 Enable
13	모든 EISC	Interrupt의 Enable
12	모든 EISC	0 : Auto-vectored Interrupt의 Enable
11	SE1608 계열	Reserved, always 0
	SE3208 이상	Extension Flag
10	SE3208 이하 계열	Reserved, always 0
	AE32000 이상 계열	Endianness, Read only. Set/clear on power-on Configuration 0 : Little Endian 1 : Big Endian
9	SE3208 이하 계열	Reserved, always 0
	AE32000 이상 계열	Porcessor mode. See bit 15
8	SE3208 이하 계열	Reserved, always 0
	AE32000 이상 계열	멀티프로세서 환경에서 버스 제어권이 반환되지 않기 위해 사용 Lock Flag
7	모든 EISC	Carry Flag
6	모든 EISC	Zero Flag
5	모든 EISC	Sign Flag
4	모든 EISC	Overflow Flag
3-0	모든 EISC	Reserved, always 0

[표 4-16] EISC 상태 레지스터



[그림 4-2] EISC 프로세서의 GPR과 비트 수

상태 레지스터의 내용은 EISC 프로세서의 종류별로 약간씩 다르며 그 비트의 크기 또한 다르나 16번 이상의 비트 자리는 사용되지 않는다. 자세한 레지스터의 내용은 각 프로세서별로 스펙을 살펴보길 바라며 대략의 내용은 위의 표와 같다. 이렇듯 다양한 레지스터들은 함수나 변수를 호출하고 값을 되돌릴 때 유용한 정보들을 가지고 동작하게 된다.
EISC SE1608, AE3208계열은 C언어의 Sub-Function으로 인자를 전달해 주거나 반환 값을 되돌려 받기 위해서 사용하는 레지스터로서 R0, R1을 사용하며 AE32000, 64000계열처럼 일반 레지스터의 개수가 8개 이상인 것은 보통 R8과 R9을 사용한다.
R6은 C/C++에서 프레임 포인터 레지스터로 사용되며, R7은 C언어에서 인덱스 레지스터로 사용된다.

4.4.3 함수 호출

C언어에서 함수 호출을 하게 되면 Sub-Function으로 인자를 주고 반환 값을 받아야 한다. 이러한 변수 또는 값들의 교환은 스택과 레지스터를 이용하여 이루어진다.
GPR(General Purpose Register)는 보통 아무런 제한 없이 사용할 수 있다. Sub-Function으로 인자를 전달해 주거나 반환 값을 되돌려 받기 위해서 사용하는 레지스터는 EISC에서 보통 R8과 R9이다. 이 두 레지스터에 차례로 변수를 저장하고 함수를 호출한다. 그러나 만일 Sub-Function에 전달할 값이나 인자가 두 개 이상이 되면 먼저 저장된 값이나 인자를 스택에 Push하고 Sub-Function을 호출하게 된다.
호출된 Sub-Function은 Return할 값을 레지스터 R8과 R9에 저장하고 호출한 함수로 되돌아간다. 만일 이 두 레지스터에 저장할 수 있는 값보다 큰 데이터를 반환해 주어야 할 경우에는 Sub-Function을 호출하기 전에 미리 할당된 스택에 값을 저장한다. 따라서 큰 값을 Return할 때는 Sub-Function을 호출하기 전에 미리 스택에 충분한 공간을 할당한 뒤 Sub-Function을 호출 해야 한다.

① 두 개의 인수를 전달하고 한 개의 인수를 되돌려 받는 경우

아래와 같은 C 소스 코드를 작성해보자. 아래의 Sub-Function은 한 개의 문자형 변수와 한 개의 정수형 변수로 정의된 인자를 받아들여 그 두 개의 값을 더하여 그 결과 값을 반환하게 된다.

```
#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;
}
```

[표 4-17] 값의 전달 및 반환에 대한 첫 번째 예제

[표 4-17] 프로그램을 보면 메인 함수에서 Sub-Function을 호출할 때 인자를 Sub-Function으로 Call by Value로 전달해 주며 Sub-Function에서 더해진 값은 반환 함수를 통해서 결과적으로 메인 함수의 값으로 반환된다. 이 프로그램을 다음과 같이 Disassemble해보자.

```
$ 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
c0700168:       60 b0           push    %R5, %R6
c070016a:       d6 e1           lea     ( %SP    ) %R6
c070016c:       ff b6           lea     ( %SP   0xFFFFFFFC     ) %SP
c070016e:       56 19           st      %R9     , ( %R6   + 0x14 )
c0700170:       02 40           leri    0x2     (0x2)
c0700172:       06 38           stb     %R8     , ( %R6   + 0x10 )

c0700174 <.LM4>:
        int result;
        result = one +  two;
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>:
}
c0700184:       c6 e1           lea     ( %R6     ) %SP
```

```
c0700186:       60 b1           pop     %R5, %R6
c0700188:       40 b5           pop     %pc
```

c070018a <_main>:
int main(void)
{

```
c070018a:       20 b4           push    %lr
c070018c:       60 b0           push    %R5, %R6
c070018e:       d6 e1           lea     ( %SP    ) %R6
c0700190:       fa b6           lea     ( %SP   0xFFFFFFE8      ) %SP
```

c0700192 <.LM8>:
        char one;
        int two, return_value;
        one = 1;

```
c0700192:       96 e4           lea     ( %R6      ) %R9
c0700194:       df c9           addq    0xffffffff,         %R9
c0700196:       01 a8           ldi     0x1     %R8
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:
```

[표 4-18] [표 4-17]의 Source Code를 Disassemble한 결과 (AE32000)

[표 4-18]에서 보면, Main 함수는 Sub_Function함수에 전달할 변수 one과 two를 각각 Register R8과 R9에 저장하고, Sub_Function함수를 호출한다. 그리고 Sub_Function함수는 Return할 RESULT변수를 Register R8에 저장하고 Main함수로 Return된다.

② 세 개의 인수를 전달하고 구조체 변수를 되돌려 받는 경우

```
#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;
}
```

[표 4-19] 값의 전달 및 반환에 대한 두 번째 예제

[표 4-19]는 한 개의 문자형 변수와 두 개의 정수형 변수를 인자로 받고, 구조체 형태의 값들을 되돌려 주는 Sub_Function 함수를 호출한 경우의 예제 프로그램이다.

이 프로그램을 Disassemble하면 [표 4-20]과 같이 나타난다.

```
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:
```

[표 4-20] [표 4-19] 의 소스 코드를 Disassemble한 결과(AE32000)

위의 Disassemble한 결과를 보면 메인 함수에서 Sub_Function에 주는 ONE, TWO, THREE 변수들 중 ONE과 TWO는 레지스터 R8, R9에 저장하고 나머지 한 개는 스택에 저장된다. Sub_Function에서 메인 함수에 되돌려 준 구조체의 변수 값은 Sub_Function의 변수 A를 메인 함수의 변수 B로 복사하여 되돌려 준 것이다. 이것은 실제로 구조체를 갖는 첫 시작 주소의 포인터를 전달 받는 것이다. 위에서 얘기된 바와 같이 레지스터가 가질 수 있는 범위를

벗어나면 나머지 값들은 [표 4-19]처럼 스택에 저장되고 함수 호출을 한다.

③ 구조체 형태의 변수를 주고 받는 경우

```
#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;
}
```

[표 4-21] 구조체 형 변수의 인자 전달 및 반환

[표 4-21]의 프로그램은 메인 함수와 Sub_Function사이에 구조체형 변수가 인자로 이용되어 전달되고 되돌려진다. 즉, 메인 함수에서 구조체형인 TOTAL의 형태로 b라는 변수가 선언이 되며 이 b라는 구조체의 sum 멤버에게만 값이 주어지고 b 구조체는 Sub_Function의 호출 시 인자가 되어 Sub_Function으로 들어간다. 이때 Sub_Function의 연산이 다 끝나면 다시 연산

된 결과의 반환 값은 b 구조체로 들어오게 된다. Sub_Function는 몇몇 간단한 연산을 수행하고 이 값을 되돌리며 메인 함수의 끝에 선언된 것과 같이 연산이 끝난 b구조체의 멤버인 sum이 메인의 대표 값으로 반환된다.

이 소스 코드의 Disassemble한 결과를 살펴보자.

```
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 (0x7FFFFF)
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:
```

[표 4-22] [표 4-21]의 소스 코드를 Disassemble 한 결과(AE32000)

위에서 보면 메인 함수에서 문자나 정수형 변수는 레지스터와 스택에 직접 값을 저장하여 Sub_Function으로 전달되어지고, 구조체 형 변수는 그 포인터만(주소 값)을 Sub_Function에 전달한 것을 알 수 있다. Sub_Function에서 메인으로 반환 할 값들은 스택영역으로 복사되어 메인 함수로 전달되었다.

4.5 Special purpose register handling

4.5.1 Program Counter (PC)

PC 레지스터는 현재 실행되고 있는 instruction의 VMA값을 가지고 있다. Assembly code programmer가 PC 값을 가지고 오기 위해서는 다음과 같은 트릭 코드가 필요하다.

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)

JAL 혹은 JALR을 실행 시키면 LR값은 현재 PC 값의 + 2 의 value를 가지게 된다. LR값을 가져오기 위해서는 다음과 같은 코드가 필요하다.

Code list – Get LR value

```
        push LR
        pop %R7 # R7 has LR value
```

4.5.3 Extension Register (ER)

ER 레지스터는 LERI 명령어를 수행하면 update 된다. ER 레지스터의 값은 status register의 E-flag field가 set되어 있는 경우에만 유효하며, assembly programmer는 ER값을 직접 handling 할 필요가 없다.

4.5.4 Multiply Result Register (MH, ML)

Multiplication 이나 Mac 연산을 수행한 결과 값은 MH/ML 레지스터에 저장된다.

Code list – Handling multiply result register

```
        MTML %r0        /* R0 register의 값을 ML register로 move 한다. */
        MTMH %r1        /* R1 register의 값을 MH register로 move 한다. */
        MFML %r2        /* ML register의 값을 R2 register로 move 한다. */
        MFMH %r3        /* MH register의 값을 R3 register로 move 한다. */
```

```
        MUL %R0, %R1
        MFML %R1        /* R0 * R1 의 결과 값을 R1으로 가지고 온다. */
```

4.5.5 Count Register (CR0, CR1)

Auto-increment를 지원하기 위한 special register로써, AE32000 architecture에 CR0, CR1 2개의 counter register가 있다. MTCR0, MTCR1 명령어는 각각 CR0와 CR1 register에 값을 setting하고, MFCR0, MFCR1은 CR0, CR1 register에서 값을 가지고 general register로 값을 move 한다. Auto-increment mode는 hardware developer guide를 참조하기 바란다. 아래의 예는 normal mode에서 auto-increment 사용예이며, libc의 memmove함수의 일부분이다.

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

- Load/Store연산에서 base 레지스터가 stack pointer register인 경우는 아래와 같이 두 가지 경우만 가능하다.

Code list – Usage load / store operation when base register is SP

```
opcode (%SP, x), %rd    /* x is constant */
opcode (%SP), %rd
```

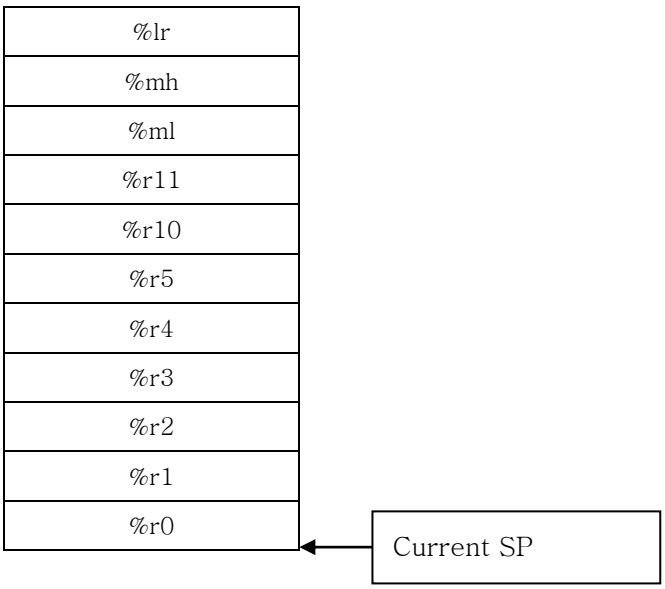
③ Push / Pop

- 레지스터를 스택에 push 및 pop 하는 명령어이다.

Code list – Usage push / pop operation

push %ml, %mh, %lr	pop %r0 - %r5
push %r10, %r11	push %r10, %r11
push %r0 - %r5	push %ml, %mh, %pc

- Frame layout of the example push code



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 연산은 3가지 type이 있다.
- 단, addq, cmpq 연산은 2번째 type만 가능하며, 3번째 type은 extb, exts, cvb, cvs ,neg, not 연산에서만 가능하다.

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

- Conditional branch의 경우는 아래의 code list에서 1,2 번째의 경우만 가능하다.

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

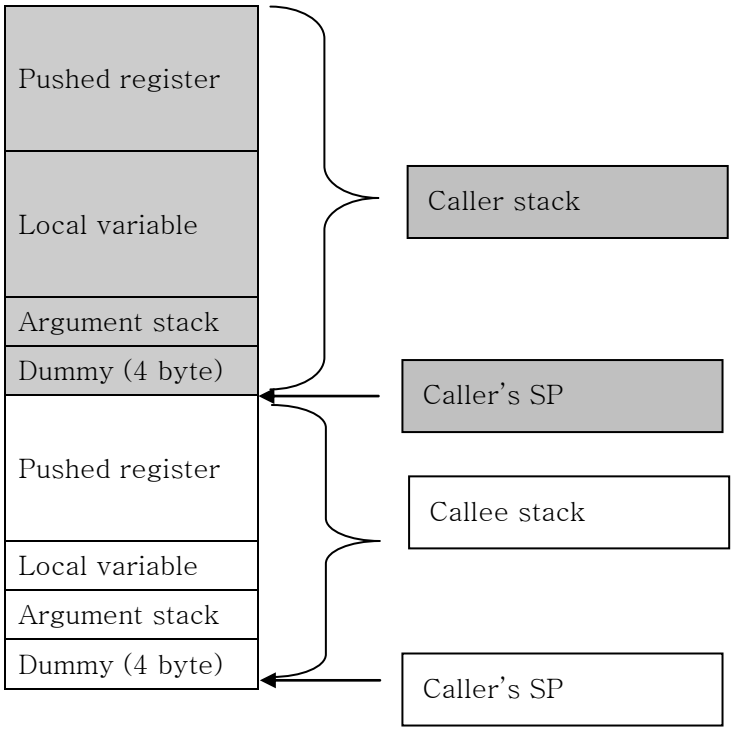
Opcode	Description
mac	Multiply and add operation

Code list – mac operation

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

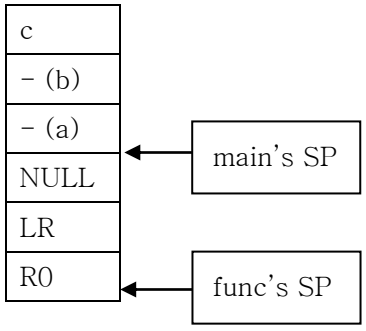
4.7 ABI (Application Binary Interface)

4.7.1 Frame layout



- 함수의 frame size는
  (# pushed register * 4 + local variable size + # max argument * 4 + 4) byte 이다.
- Pushed register는 함수의 시작지점에서 callee에서 사용하기 위한 register를 compiler가 저장하는 register들이다.
- Local variable size는 compiler가 결정하며, -O0 option으로 compile 하였을 때, 그 함수 안에서 정의된 모든 local variable의 크기의 합이다. 그러나, -O2 option으로 compile 하면 register allocation routine으로 들어가기 때문에 local variable의 size는 spill된 register의 size와 동일하다.
- Max argument stack size는 함수에서 호출하는 callee들 중 argument들의 size를 비교하여, 가장 큰 크기를 자신의 argument size로 결정한다.
- Argument register는 R8, R9가 있으며, 각각 첫 번째, 두 번째 actual parameter를 전달한다. structure type을 argument로 넘길 때는 structure의 data가 저장된 첫번째 member의 주소를 전달한다.

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
		main 함수의 스택 사이즈는 20byte 이며, 이것은 4(pushed lr), 16(Max argument size + dummy)의 합으로 계산되었다. main 함수에는 지역변수를 위한 스택이 할당되어 있지 않다.
		
		func 함수는 8byte의 스택 사이즈를 갖는데, 이것은 push된 레지스터의 크기만 고려된 것이다. func 함수에서 세 번째 argument 'c'를 가지고 오기 위해서 ld (%sp, 20), %r0와 같이 ld 명령어를 사용하였으며, 20은 8(func stack size)과 3*4(argument size)의 합이다.
		
		 

- AE32000 EISC compiler version 2.7은 Global pointer register가 구현되어 있으며, R7 register는 data section의 start point VMA의 값을 가지고 있다. 그러므로, assembly programmer는 R7값을 사용할 때에 주의 하여야 한다.

4.8 built-in functions

4.8.1 void * __builtin_return_address(unsigned int LEVEL)

- __builtin_return_address는 현재 함수의 return address 값을 가지고 온다. argument로는 0만 허용되며, 현재 함수의 return address를 알고자 할 때 사용한다.

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)

- __builtin_frame_address 함수는 현재 함수의 frame point의 위치를 가지고 온다. argument는 0만 허용된다. Function의 시작은 callee saved register의 push 후 local stack area의 할당 순이다. Frame point의 위치는 callee saved register의 push가 끝난 후의 stack pointer의 위치이다. 즉 current stack point + local stack size가 이 함수의 return value이다.
-

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)

- TYPE1과 TYPE2의 호환성을 검사한다. AE32000에서 data type은 서로 호환하지 않기 때문에 TYPE1과 TYPE2가 다르면 0을 return하고 같으면 1을 return 한다.

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 4.8.4 int __builtin_constant_p(EXP)

- Argument인 expression이 constant인지 판별한다. Constant이면 1을 return하고 아니면 0을 return 한다.

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)

- Positive infinity를 return 한다.

4.8.6 float __builtin_huge_valf(void)

- Positive infinity를 return 한다.

4.8.7 Additional built-in functions

- 이 외에도 AE32000 GCC compiler는 ISC C, ISO C99, ISO C90 built-in 함수를 지원한다.

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

5 장

Exception Handling

Exception은 프로그램에 의하여 수행되는 정상적인 프로그램의 흐름을 방해하는 모든 상황을 의미한다. 여기서는 다음 단원이 포함되어 있다.

- 프로세서의 Exception프로세서의 Exception
- Exception의 종류Exception의 종류
- 인터럽트 벡터 테이블Exception Vector 테이블.
- Vector Base

- Exception Priority
- Exception Service Routine 진입
- Exception Service Routine 종료
- Exception 처리 과정

5.1 프로세서의 Exception

Exception은 Reset이나 인터럽트와 같은 CPU와 비동기적인 예외상황을 처리하기 위한 신호이다. 이 신호에는 System의 동작에 있어서 요구되어지는 특별한 처리를 행하는 신호와 응용 프로그램의 동작에 필요한 처리를 행하기 위한 신호가 있다. 각각의 Exception들은 각각의 처리 프로그램들과 연결되어지는데 그 방법은 Exception의 종류에 따라 다르다. 그리고 각 Exception간에도 우선 순위가 있어서, 현재 수행중인 Exception의 우선 순위보다 높을 경우에만 새로운 Exception의 수행이 허용된다. 만일 현재 수행되고 있는 Exception의 우선 순위보다 낮을 경우에는, 현재 수행되는 Exception의 수행이 종료되어야만 새로이 발생한 Exception을 수행할 수 있다.

CPU에는 User Mode, Supervisor Mode, OSI Mode가 있는데, Exception이 발생하게 되면 Supervisor Mode로 전환되어 Exception처리 프로그램을 수행한다. 단 OSI Exception일 경우에는 OSI Mode로 전환되어 OSI 프로그램을 수행한다. 이러한 CPU Mode는 SR(Status Register)의 Bit들(예 AE32000에서는 bit15 와 bit9)에 의해서 결정되고, Exception이 발생하면 CPU에서 자동으로 Mode를 전환하게 된다. 그리고 Exception처리가 끝나면 Stack Area에 PUSH했던 SR을 다시 POP하므로 자동으로 이전 Mode로 전환된다. OSI Mode는 CPU에 따라 존재할 수도 있고 없을 수도 있다. 그리고 OSI Mode가 존재하면 ISP(OSI Stack Pointer)도 존재한다.

5.2 Exception의 종류

5.2.1 Reset

코어 프로세서의 초기화 동작을 요구하고자 할 때, 외부에서 코어 프로세서로 입력해 주는 신호를 통하여 발생한다. 이 인터럽트를 받을 경우 프로세서는 초기화를 수행한다. 초기화 시 모든 범용 레지스터의 값이 알 수 없는 값 또는 '0'의 값(구현 모델에 따라 변경될 수 있으므로 technical reference manual을 참조할 것)으로 변경된다. PC의 값은 리셋 벡터의 값으로 변경되며, SR은 '0'이 된다. LR,ER, MH, ML, SSP, ISP, USP의 값의 초기화는 구현에 따라 '0'또는 'unknown'값이 된다.

5.2.2 External Hardware Interrupt

외부 하드웨어 모듈에서 요청하는 인터럽트를 의미한다. 현재 상태를 관리자 스택에 PUSH한 후 인터럽트 핸들러 주소로 PC값이 이동한다. 현재 상태들 중 PC, SR값이 관리자 스택으로 PUSH되며, 다른 PUSH가 필요한 레지스터들은 인터럽트 핸들러에 의하여 관리되어야 한다. 외부 인터럽트는 다음 두 가지 방식으로 인터럽트 벡터를 지정할 수 있다.

- Autovectored Interrupt : 인터럽트 벡터가 지정되어 있는 경우로서, 인터럽트가 발생한 경우 내정되어 있는 인터럽트 벡터로 이동한다.
- Vectored Interrupt : 외부 인터럽트 컨트롤러에서 해당 인터럽트의 번호를 넘겨줌으로써 인터럽트 벡터의 위치를 판별할 수 있는 경우

5.2.3 Software Interrupt

소프트웨어적으로 발생시키는 인터럽트를 의미한다. 관리자의 자원을 사용자 모드에서 접근하고자 하는 경우 호출한다. 일반적으로 system call을 위하여 사용된다. 현재 프로세서의 상태를 PUSH하고, 지정된 인터럽트 번호로 지정되어 있는 인터럽트 벡터로 이동한다.

5.2.4 Non-Maskable Interrupt

Non-maskable interrupt(NMI)는 인터럽트 컨트롤러에서 마스킹(인터럽트 요청을 거부하는 것) 동작을 할 수 없는 인터럽트를 의미한다. 일반적으로는 메모리 접근에서 패리티가 발생한 경우들에서 사용되지만, AE32000에서는 시스템 보조 프로세서에서 이러한 상황에 대한 인터럽트를 발생시키므로(시스템 보조 프로세서 인터럽트 참조) 외부에서 직접적으로 입력되는 특정 인터럽트와 연결한다. AE32000에서 NMI는 인터럽트 컨트롤러를 거치지 않고 직접 입력되는 인터럽트로서 사용된다.

5.2.5 System Coprocessor Interrupt

시스템 보조 프로세서 인터럽트는 메모리 접근 과정에서 발생하는 인터럽트들을 의미한다. 예를 들자면 관리자 영역으로 지정된 부분을 사용자가 접근 하거나, 목적 주소에 메모리가 존재하지

않거나, 메모리 read/write과정에서 패리티 오류가 발생한 경우 모두 시스템 보조 프로세서 인터럽트가 발생한다.

5.2.6 Coprocessor Interrupt

보조 프로세서 인터럽트는 보조 프로세서의 접근 과정에서 발생하는 인터럽트를 의미한다. 또한, 이 인터럽트는 연산을 수행하는 보조 프로세서의 연산 결과(상태비트)를 Polling하여 발생시킬 수 있다. 보조 프로세서 접근 과정에서 발생하는 인터럽트는 관리자 전용으로 지정된 보조 프로세서를 사용자 모드에서 접근할 때 발생한다.

5.2.7 Breakpoint & Watchpoint Interrupt

OSI에서 제공하는 디버깅 기능을 제공하기 위한 인터럽트이다. OSI 모듈에서 지정된 breakpoint와 watchpoint의 조건을 만족하였을 때 이 인터럽트가 호출되어 프로세서의 모드를 OSI모드로 변경시킨다.

5.2.8 Bus Error & Double Fault

명령어 버스나 데이터 버스의 복구 불가능한 버스의 오류인 경우에 Bus error가 발생하며, 인터럽트 핸들러 fetch혹은 상태 PUSH과정에서 오류가 발생한 경우 해당 인터럽트를 더 이상 처리할 수 없으므로 double fault가 발생한다. 이 경우 supervisor stack의 영역에 문제가 있거나, 인터럽트 핸들러의 위치를 변경시키는 Vector base register의 설정이 잘못되어 있는 경우 발생할 수 있다.

5.2.9 Undefined Instruction Exception

정의되지 않은 명령이 입력된 경우 발생하는 예외이다. 이 경우 버전에 따라 오류를 발생시키거나, NOP로 처리 할 수 있다.

5.2.10 Unimplemented Instruction Exception

명령어 셋 상에 정의되어 있으나, 해당 버전에서는 구현되어 있지 않은 명령을 의미한다. 이 경우 인터럽트를 통하여 소프트웨어적으로 에뮬레이션함으로써 값을 얻어낸다. 호환성을 높이기 위하여 사용된다.

5.3 인터럽트 벡터 테이블

5.3.1 개요

Exception의 처리를 위하여 프로세서는 인터럽트 핸들러 루틴을 읽어와야 하는데, 이 인터럽트 핸들러 루틴의 주소는 인터럽트 벡터 테이블에 존재한다. AE32000에서 인터럽트 벡터 테이블은 기본적으로 0x0번지에 존재하며, 각 인터럽트는 아래 그림과 같이 정의되어 있다. 그림에서 shade처리 되어 있는 인터럽트들은 vector base register로 이동시키는 것이 불가능하다.

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

[그림 5-1] ae32000의 인터럽트 벡터 테이블

인터럽트 벡터 테이블이란 인터럽트 핸들러 함수의 포인터로 구성된 배열로서 AE32000프로세서는 0x00000000번지에 위치한다. 인터럽트 벡터 테이블은 필요에 의하여 재배치가 가능한데 위의 회색으로 표시된 인터럽트들은 재배치가 불가능하다.

인터럽트 벡터는 앞에서 언급하였듯이 0x0번지에 위치해야 하므로, C컴파일러를 사용하는 경우에는 linker script에서 반드시 이 위치가 0x0에 올 수 있도록 memory layout을 지정하여야 한다.

```
SECTIONS
  {
  /* Read-only sections, merged into text segment: */
  . = 0x0;
  .text :
  {
    *(.vects)  /*  이 부분이 인터럽트 벡터 위치를 가리키게 된다. */
```

[표 5-1]

위의 링커 스크립트는 인터럽트 벡터를 0x0에 위치시키기 위한 것이다.

위의 ".vects"는 section을 가리키는 symbol로서, 링커에서는 이러한 section symbol을 기반으로 메모리를 배치하게 된다. 물론, 이러한 section symbol은 코드 내에 존재해야 의미를 지닌다. 다음은 ".vects"가 정의된 부분을 보도록 한다.

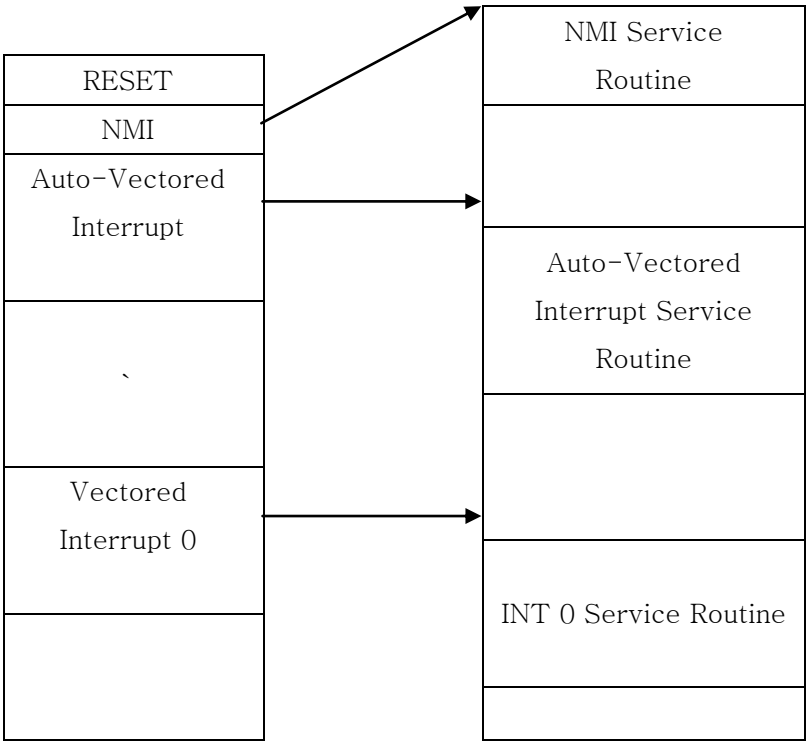
```
typedef void (*fp)(void);
const fp HardwareVector[] __attribute__((section (".vects")))= {
        start, /* V00 : Reset Vector */
        nmi, /* V01 : NMI Vector */
        auto_int, /* V02 : Interrupt Auto Vector */
        dfault_int, /* V03 : Double fault Vector */
        berr_int, /* V04 :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

[표 5-2]

위의 벡터테이블 정의에서 볼 수 있듯이 벡터 테이블은 인자를 지니지 않고, 반환 값을 지니지 않는 함수 포인터들의 배열로 이루어져 있다. 또한, 테이블의 속성으로서 section symbol을 ".vects"로서 지정하여 링커에서 이를 인식 할 수 있도록 해 두었다. 인터럽트 함수의 작성 방법에 대한 자세한 사항은 software reference guide를 참조하면 된다.

5.3.2 Exception Vector와 Exception Service Routine

아래 그림은 Exception Vector와 Exception Service Routine의 관계를 보여주고 있다. Exception Vector는 Exception Service Routine의 시작 주소를 저장하고 있는 Pointer이다. 따라서 Exception Vector를 읽어 PC에 넣으면 다음에 CPU는 PC가 지시하는 주소의 명령어를 읽어와 실행하기 때문에 Exception Service Routine을 실행할 수 있다.



[그림 5-2] 인터럽트 벡터와 서비스루틴의 관계

Exception Service Routine은 입력과 출력을 가질 수 없다. 즉 Void형을 갖는다. 따라서 다음과 같은 구조를 가진다.

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

[표 5-3]

AE32000 compiler에서 지원하는 interrupt function은 두 가지 경우가 있으며, 각각 #pragma interrupt, #pragma interrupt_fast로 구분된다. #pragma directive는 컴파일러에게 interrup 함수임을 알려주며, 컴파일러는 interrupt 함수의 prologue와 epilogue에 현재의 state를 store 및 roll-back하는 명령어를 생성한다.

interrupt 함수의 시작부분에는 모든 general register를 push하고, cr0, cr1, ml, mh, er, lr의

6 개의 레지스터를 push한다. 즉, interrupt가 호출 되기 전의 process의 상태를 저장한다. 그리고, interrupt 함수의 종료 시에는 모든 general register를 pop하고, cr0, cr1, ml, mh, er, lr, pc, sr의 8개의 레지스터를 pop함으로써 이전 process의 상태를 되돌린다. pc와 sr레지스터는 하드웨어에서 interrupt 함수를 시작하기 전에 메모리에 저장하므로, interrupt 함수 내에서 push할 필요가 없으며, 단 interrupt 함수 종료시에는 pop을 통하여 원래의 값으로 되돌려야 한다.

interrupt_fast함수는 위에서 설명한 interrupt 함수와는 다르게 general register를 저장하지 않는다. general register를 사용하지 않는 interrupt의 경우 interrupt_fast directive를 사용하여, 실행속도를 더 빠르게 할 수 있다.

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

#pragma interrupt_fast
void INTERRUPT_FUNCTION_FAST(void)
{
  … ..
  Exception service 수행
  … ..
}
```

```
        .global _func
        .type    _func, @function
_ INTERRUPT_FUNCTION:
        push %ml , %mh , %er , %lr
        .short   0xb403    # push   %cr0, %cr1
        push %R8, %R9, %R10, %R11, %R12, %R13, %R14, %R15
        push %R0, %R1, %R2, %R3, %R4, %R5, %R6, %R7
        lea (%sp,-20),%sp          # sp=sp+ i
        mov %r8,%r4
        ……..
        mov %r8,%r0
        add %r9,%r0
        st %r0,(%sp,12)
        lea (%sp,20),%sp           # sp=sp+ i
        pop %R0, %R1, %R2, %R3, %R4, %R5, %R6, %R7
        pop %R8, %R9, %R10, %R11, %R12, %R13, %R14, %R15
```

```
        .short 0xb503    # pop    %cr0, %cr1
        pop %ml, %mh, %er, %lr, %pc, %sr
        .size    _func, .-_func
        .align 1
        .global _func2
        .type    _func2, @function
_ INTERRUPT_FUNCTION_FAST:
        push %ml , %mh , %er , %lr
        .short   0xb403    # push   %cr0, %cr1
        lea (%sp,-20),%sp          # sp=sp+ i
        mov %r8,%r4

        lea (%sp,20),%sp           # sp=sp+ i
        .short 0xb503    # pop    %cr0, %cr1
        pop %ml, %mh, %er, %lr, %pc, %sr
        .size    _func2, .-_func2
```

[표 5-4]

위의 [표 5-4]는 interrupt와 interrupt_fast directive를 사용한 C code의 예이며, 컴파일 한 결과 assembly code를 보여준다. 컴파일 결과인 assembly code를 보면, Register들의 값을 스택에 저장하고 Exception Service가 끝나면 스택에 저장했던 Register의 값들을 다시 복귀시키고 Exception이 발생하기 이전으로 되돌아 간다.

SWI는 Register R8을 통하여 값을 받아들이다. 이런 경우 SWI를 호출하기 전에 입력 값을 R8에 SWI로 전달한 값을 저장하고 SWI를 호출한다. SWI에서는 입력 값을 읽어 적절한 처리를 하고 SWI Service Routine 후 본래 실행 중이던 프로그램으로 되돌아 온다.

```
___trap0:
        swi 0x6
        leri     0x0
        cmp 0x0,%R8
        jz   4a42 <.L0>
        leri     0x3000
        leri     0x88
        ldi 0xc0000884,%R7
        st   %R8,(%R7,0x0)
        jplr
```

[표 5-5]

이렇게 작성한 Exception Service Routine을 Exception Vector Table에 연결시키기 위해서는 다음과 같이 Exception Vector Table에 Exception의 순서에 따라 함수의 이름을 적어주면 된

다. 이때 함수는 입 출력이 모두 void로 정의되어야 한다.

```
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)
{
… ..
NMI exception service 수행
… ..
}
#pragma interrupt
void auto_int_serv (void)
{
… ..
Auto-vectored interrupt service 수행
… ..
}
```

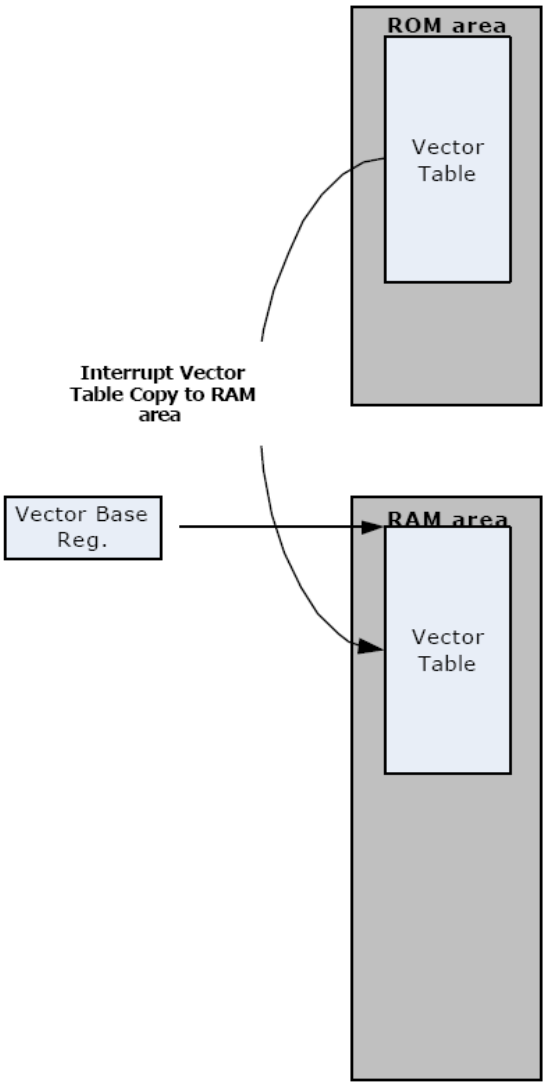
[표 5-6]

5.4 Vector Base

5.4.1 Vector Base

벡터 베이스란 Interrupt vector table의 위치를 의미한다. 기본적으로 인터럽트 벡터 테이블은 0x0번지에 위치하며, inexact exception processing을 수행하는 몇 가지 인터럽트들의 벡터 위치는 변경 할 수 없다 (이동될 수 없는 벡터들은 위의 [그림 5-1]의 벡터테이블 표에 shade로 지정되어 있으므로, 이를 참조할 것). 벡터 베이스는 시스템 보조 프로세서의 벡터 베이스 레지스터에 저장되며, 이 값이 지정된 이후부터 대부분의 인터럽트는 변경된 인터럽트 벡터 테이블로부터 벡터 주소를 읽어온다. 벡터 베이스 레지스터는 벡터 테이블의 접근을 ROM영역에서 RAM영역으로 변경함으로써, 동작의 수행 시간을 단축시키고 인터럽트의 hooking을 가능하게 할 수 있다.

Vector base에서 지정된 벡터 테이블 영역은 가상 메모리를 사용하는 경우에 반드시 TLB(Translate Look-aside buffer)내에 존재해야 한다. 이는 AE32000의 경우 software - managed TLB를 채용하고 있으므로, 벡터 처리 과정 중의 벡터 테이블 접근에서 TLB miss exception이 발생하는 경우 불필요한 double fault가 발생할 수 있기 때문이다.



[그림 5-3] 벡터 베이스

인터럽트 벡터 테이블을 재배치하는 경우 vector base레지스터로 재배치된 인터럽트벡터 테이블의 위치를 지정할 수 있다. vector base레지스터가 지정되어 있는 경우, 인터럽트가 발생하면, vector base레지스터를 기반으로 벡터테이블에 접근한다.

5.5 Exception Priority

Lucifer 프로세서의 인터럽트는 우선 순위를 지닌다. 프로세서 내부적으로 지니고 있는 우선 순위는 다음과 같다.

순위	설명	약어
1	Reset	RST
2	Bus Error	BERR
3	Double Fault	DF
4	OSI Exception	OSI
5	Coprocessor Exeception	CP
6	Sysstem-Coprocessor Exception	CP0
7	Non_maksable interrpupt	NMI
8	Software interrupt	SWI
9	Interrupt	INT
10	Halt	HALT
11	Undefined Instruction Exeception	UDI
12	Unimplemented Instruction Exception	UII

[표 5-7] 인터럽트 우선 순위

인터럽트 간의 우선순위는 '같은 시점에서 발생한 두 인터럽트'의 우선 순위를 가리는데 사용되며, 일반적으로는 먼저 요청된 인터럽트가 우선한다.

5.6 Exception Service Routine 진입

CPU는 Exception이 발생하여 현재 발생한 Exception보다 더 우선순위가 높은 Exception이 처리 중인지 확인하고 만일 더 높은 우선 순위의 Exception이 수행되고 있지 않다면, Exception 처리를 위해 다음과 같은 과정을 수행한다.

1. 현재 수행하고 있는 프로그램을 Exception후에 계속 수행할 수 있도록 SR과 PC 값을 스택에 저장한다.
2. SR(Status Register)의 값을 바꾸어 적절한 CPU Mode로 전환한다.
3. 해당 Exception Vector에 저장된 Exception Service Routine의 주소를 읽어들여 PC(Program Counter)에 넣는다.
4. PC 가 지정하는 주소의 명령어를 읽어 들여 수행하기 시작한다.
5. Exception Service Routine을 수행한다.

이때 Exception이 발생하기 전의 Register값을 저장하는 스택으로 OSI Exception일 경우에는 OSI Stack(ISP로 지정)을 사용하고, 그 외의 Exception이 발생하였을 경우에는 SSP(Supervisor Stack Pointer)를 이용하여 Supervisor Stack에 저장한다. 스택에 Register값을 저장할 때 (PUSH 할 때) SP(Stack Pointer)는 Register값을 저장하고 SP를 Register의 Byte수 만큼 감소시키게 된다. 반대로 스택에서 값을 읽어 Register에 저장할 때에는 SP를 먼저 증가 시키고 스택에서 값을 읽어내어 Register에 저장한다. 만일, HALT 상태에서 이러한 Exception이 발생하면, SWI와 Double Fault를 제외한 나머지 Exception은 HALT 상태에서 벗어나게 되다.

아래는 AE32000의 경우 Exception들이 Exception Service Routine을 수행하기 위한 동작 과정을 보여주고 있다.

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

[표 5-8]

5.7 Exception Service Routine 종료

Exception Service Routine을 종료하고 Exception이 발생하기 전 프로그램으로 복귀하기 위해서는 스택에 저장한 Register의 값을 복원시켜야 한다. 스택에서 POP 할 때에는 스택에 PUSH한 순서와 반대로 한다. 일반 Register를 모두 POP하고 마지막으로 SR과 PC를 POP하면 Exception처리 과정이 종료되고 Exception이 발생하기 전 프로그램으로 복귀하게 된다.
AE32000의 경우 SR을 PC보다 먼저 스택에 PUSH했으므로 PC를 먼저 POP해야 한다. 하지만 이 동작을 두 개의 명령어를 사용하여 "POP %PC", "POP %SR"을 실행하게 되면 PC를 POP하면서 Exception이 발생하기 이전의 프로그램으로 복귀하므로 "POP %SR"명령을 실행하지 않게 된다. 따라서, PC와 SR을 POP할 때에는 반드시 하나의 명령어를 사용하여 "POP %PC. %SR"과 같이 해야 한다.
다음은 Assembly 언어로 작성한 Auto-Vectored Interrupt Service Routine 의 Register 값의 Push와 Pop의 한 예이다.

```
_auto_int:
push %lr,%er
push %r0-%r3 /* 사용할 Register 만 Stack에 저장 */
… …
pop %r0-%r3
pop %lr,%er
pop %pc,%sr /* 이전 프로그램 으로 복귀 */
/* end auto-vectored interrupt service routine */
```

[표 5-9]

다음은 C 언어로 작성한 Auto-Vectored Interrupt Service Routine의 Register값의 Push와 Pop의 한 예이다.

```
#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
        “);
}
```

[표 5-10]

또한 SWI를 사용할 경우에는 SWI를 호출한 프로그램과 SWI간에 Data교환이 이루어지기도 하므로 두 프로그램간의 Data교환 규약을 확인하여 PUSH와 POP을 사용해야 한다. C 언어로 Exception Service Routine을 작성하면 C 함수에서 자동으로 SR과 PC를 POP해준다.

5.8 Exception 처리 과정

5.8.1 Reset

CPU의 Reset pin이 Enable되면 CPU는 수행중인 모든 작업을 중단하고 CPU를 초기화 한다. Reset이 수신되면 모든 작업을 즉시 중단하고 Exception처리 과정으로 전환된다. AE32000에서는 Status register Clear하고 OSI의 enable 상태 여부에 따라 리셋 벡터를 가져오게 된다. OSIEN과 OSIROM이 둘 다 Enable 되어 있으면 0xFFFF_0000에서 리셋 벡터를 가져오고 OSIEN만 enable 되어 있으면 0x0000_0030에서 리셋 벡터를 가져온다.

5.8.2 NMI

NMI핀으로부터 발생하는 외부인터럽트로서 Masking이 불가능한 인터럽트를 의미한다. NMI 인터럽트는 프로그램에 수행에 반드시 필요한 중요한 I/O의 처리 등에 사용된다. NMI가 발생하면 Status Register를 AE32000은 SSP(Supervisor Stack Pointer)를 이용하여 스택에 저장한다. 그런 후 SE3208에서는 NMI, Interrupt를 disable시키고 extension flag를 clear 시킨다. AE32000에서는 여기에 추가적으로 supervisor 모드 상태로 진입하기 위해 processor mode를 clear 시킨다. 그런 다음 PC값을 스택에 저장하고 메모리의 4번지에서 NMI Vector를 PC로 가져와서 NMI 처리 프로그램을 실행한다.

5.8.3 SWI

SWI는 프로그램에 의해서 발생하게 된다. EISC 명령어 중 SWI 명령에 의해서 발생한다. SWI가 발생하면 Processor Mode는 Supervisor Mode로 전환된다. SWI가 발생하면 Status Register를 AE32000은 SSP(Supervisor Stack Pointer)를 이용하여 스택에 저장한다. 그런 후 NMI, Interrupt를 disable시키고 extension flag를 clear 시킨다음 여기에 추가적으로 supervisor 모드상태로 진입하기 위해 processor mode를 clear 시킨다. 그런 다음 PC값을 스택에 저장하고 SWI 벡터 번호에 따라서 SWI Vector를 PC로 가져와서 NMI 처리 프로그램을 실행한다.

5.8.4 Interrupt

Interrupt는 Status Register의 bit 12가 '0'이면 Auto-Vectored Interrupt로 동작하고 '1'이 Vectored Interrupt로 동작한다. Auto-Vectored Interrupt와 Vectored Interrupt의 차이점은 Interrupt vector를 가져오는 위치이다. Auto-Vectored Interrupt는 메모리의 8번지에서 Exception Vector를 가져오고, Vectored Interrupt는 CPU외부에서 Exception Vector를 수신하여 메모리의 주소를 계산하여 Exception Vector를 가져온다.

Interrupt가 발생하면 Status Register를 AE32000은 SSP (Supervisor Stack Pointer)를 이용하여 Stack에 저장한다. 그런 후 NMI, Interrupt를 disable시키고 extension flag를 clear 시킨다음 supervisor 모드상태로 진입하기 위해 processor mode를 clear 시킨다. 그런 다음 PC값을 Stack에 저장하고 PC에 해당 Interrupt Vector를 가져와 저장하여 Interrupt처리 함수를 실행한다.

5.8.5 OSI Break Exception

OSI는 Chip내부에 내장되어 있는 Debugging Module이다. 실제로 OSI는 내부의 Debugging 정보를 모두 메모리로 Dump하는 형식으로 Debugging이 진행되어야 하지만 실제 그러한 구현을 하기 위해서는 만만치 않은 Cost가 들어가야 하므로 약식의 Debugging Module이 들어가게 된다. 주의해야 할 점은 OSI는 특별하게 통신 Channel을 정의하지 않는다는 점이다. 즉, UART이건 단순한 Serial 통신이건 간에 같은 방식으로 운용 할 수 있다는 점이다. 최대 8개의 Break Pointer Comparator를 둘 수 있다(8개 Channel). OSI (On Silicon ICE) Break은 외부 OSI Block에서 생성해 주는 것으로 명령어 Fetch시 사용된 주소, 또는 Data접근 시 사용된 주소와 OSI Breakpoint와 비교하여 두 값이 같다면 Break를 수행한다.
예로서 AE32000에서는 OSI Break을 지원하기 위하여 BRKPT명령어가 존재하며, 이 명령은 OSI Mode에서 동작하면서 OSI Break Exception을 발생시키는 역할을 수행한다. 명령어 주소에 의한 OSI Break의 경우 명령이 BRKPT로 치환 되는 것이므로 OSI break Exception을 발생시킨 명령어는 수행되지 않는다. 이 OSI에 대한 좀 더 자세한 설명은 Debug manual에서 설명되어진다.

5.8.6 Co-Processor Exception

Co-Processor의 동작에 영향을 받아 어떤 동작을 수행하기 위해서 사용하는 예외 처리 상황이다. 이것을 수행하면 특정 상태 bit를 받아 오게 되므로 Cp-Interface에서는 해당 Address를 발생시키게 된다. 이때 Address Bus와 CP Number Bus를 공유하게 되면 충돌이 일어나므로 별도의 Bus를 할당한다.

5.8.7 Bus Error Exception

System의 Bus에서 Error가 발생되어서 그 것을 Core에 알려주는 경우에 해당된다. 즉, 지정한 주소가 없는 주소이거나 Data가 손상된 경우를 의미하며, Memory Controller로부터 발생되는 신호이다. 이런 경우 대개는 Write Buffer에 Data가 저장되어 있고 그 시간 동안 Core가 상당히 진행되었으므로 언제 Error가 나는지에 대한 정확한 정보가 없다. 따라서 이 경우 복구할 수 없는 Error를 발생시키므로 모든 것을 포기하고 새로 시작해야 한다.
CP0 (Co-Processor 0)가 있는 경우 CP0는 Memory Controller의 Bus Error 신호를 받아 Data 접근 과정에서의 Error인지, 명령어 접근 과정에서의 Error인지를 판별하여 dberr/iberr 신호를 Core 에 공급한다. CP0가 없는 경우 Memory Controller의 Bus Error 신호는 Core의 dberr/iberr 모두에 연결되며 접근명령(Load/Store) 수행 중인 경우 dberr이 발생한 경우와 동일하게 처리하고, Data 접근 명령 수행 중이 아닌 경우 iberr과 동일하게 처리한다.