Lucida User Guide

AE32000C-Lucida Processor

Processor Team, R&D center,
ADChips Inc.

Revision : 2.0
November 9, 2011

Document revision histroy

```
$Log: lucida-userguide-ko.tex,v $
Revision 2.0  2011/10/25 18:17:00  eundol
Renewal
```

Lucida User Guide: AE32000C-Lucida Processor

Office
22th Floor, KeomKang Penterium IT Tower Bldg., 810, Gwanyang-dong, Dongan-gu, Anyang-si, Gyeonggi-do, 431-060, Korea.
Tel : +82-2-2107-5800
Fax : +82-2-571-4890
URL : http://www.adc.co.kr

Contents

List of Tables

List of Figures

This page intentionally left blank

TRADEMARKS

EISC®는 Advacned Digital Chips Inc.의 등록상표입니다.
AE32000®은 Advacned Digital Chips Inc.의 등록상표입니다.
AMBA®, AHB®, AXI®, APB®는 ARM Inc.의 등록상표입니다.
GNU toolchain은 GPL/LGPL을 따릅니다.
기타 각사의 등록 상표는 각사의 소유입니다.

Chapter 1

Introduction

이 장에서는 Advanced Digital Chips Inc.에서 개발한 32비트 EISC (Extendable Instruction Set Computer) 프로세서인 AE32000C-Lucida에 대한 전반적인 사항을 알아보고, 본 문서를 사용하는 방법에 대하여 다루도록 한다.

1.1 EISC microprocessor

1.1.1 EISC processor overview

EISC(Extendable Instruction Set Computer)는 Advanced Digital Chips Inc.(http://adc.co.kr, 이후 ADChips라 표기)에서 개발한 명령어 셋 아키텍처로서, 내장형 응용 분야(embedded application)에 최적화된 명령어를 제공한다. 이를 위하여 EISC 프로세서 아키텍처는 메모리 접근에 직접적인 영향을 주는 프로그램의 크기를 줄이면서도 적절한 수의 레지스터를 제공함으로써 데이터 메모리 접근 빈도를 줄이는 방법을 사용하였다. 이러한 접근 방법은 형태상 축약 명령어를 사용하는 RISC(compressed code RISC)의 접근 방법과 매우 유사하나, EISC라는 이름에서 알 수 있듯이, `leri` 라 불리는 즉치값 확장 명령어를 이용하여 필요한 경우 명령어에서 사용되는 즉치 값(immediate value)이나 변위(offset)를 자유롭게 확장할 수 있는 형태를 가질 수 있도록 함으로써, 축약 명령어를 사용하는 RISC에서 문제가 되어 온 즉치 값 사용의 문제를 해결하였다는 장점을 지니고 있다.

EISC 프로세서는 구조적으로 RISC와 유사하므로 하드웨어 형태가 비교적 단순하다는 장점을 지니고 있으며, 명령의 확장의 방식에 있어서 CISC와 유사점이 있으므로 CISC와 필적하는 코드 밀도[1]를 보여준다는 장점을 지닌다.

1.1.2 Processor Family

EISC 프로세서는 그림. 1.1에서 나타낸 것과 같이 16비트, 32비트, 64비트 프로세서군으로 나누어지며, 32비트 마이크로 프로세서의 경우 성능을 기준으로 마이크로 프로세서의 경우 마이크로 컨트롤러 시장을 위한 SE(simple EISC) processor와 AE(Advanced EISC) processor군으로 나누어진다. 각 프로세서에 대한 자세한 사항은 프로세서 메뉴얼을 참조하면 된다.

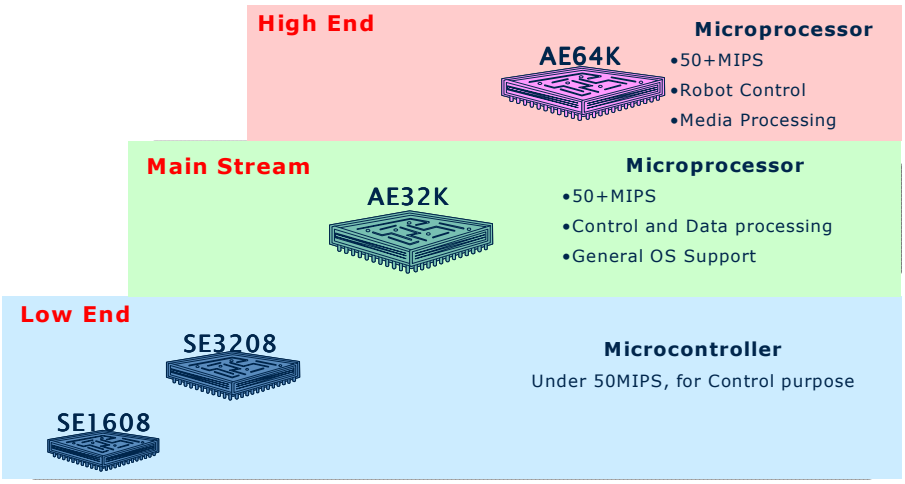


Figure 1.1: EISC processor family

[1] 코드 밀도(code density): 코드 밀도는 일반적으로 컴파일된 코드의 크기에 역수를 취하여 얻는다. 따라서, 코드 밀도가 크다는 것은 같은 프로그램을 작성하였을 때 더 작은 코드가 나온다는 것을 의미한다

1.2 AE32000C-Lucida Overview

AE32000은 EISC 명령어 셋 아키텍처를 채용한 32비트 EISC 명령어 셋 아키텍처로써, 고성능/저전력 마이크로 컨트롤러 및 다양한 ASSP(Application Specific Standard Product)에의 적용을 목적으로 하고있다. AE32000 아키텍처는 RTOS(Real Time OS)뿐만 아니라, Linux와 같이 MMU가 요구되는 일반적인 O/S를 지원할 수 있으며, 강력한 디버깅 기능을 내장하고 있다. AE32000 아키텍처는 필요에 따라 3개 까지의 특수 목적 보조 프로세서를 추가할 수 있도록 고안되어 확장이 용이하다.

AE32000 아키텍처는 초기 발표 이후에 안정성 강화 버전인 AE32000B ISA를 거쳐, SIMD-DSP 기능이 대폭 강화된 AE32000C 아키텍처로 발전되었으며, 현재 일반적으로 사용되는 AE32000기반의 프로세서는 모두 AE32000C 명령어 셋 아키텍처를 기반으로 하고 있다.

AE32000C-Lucida 프로세서는 일반적으로 AE32000C로 통칭되는 AE32000 ISA revision C를 기반으로 구현된 마이크로 프로세서이다. AE32000C-Lucida 프로세서는 Verilog HDL기반의 Soft IP형태를 취하고 있으므로, 네트웍, 신호처리, 가전 분야를 위한 ASSP(Application Specific Standard Product)를 비롯한 다양한 SoC에서 적용 가능하다.

본 프로세서는 기존의 AE32000B-Calliope 프로세서나 AE32000C-Lucifer 프로세서와 비교하여 바이너리 수준의 하위 호환성을 유지하면서, 신호 처리 전용 명령어를 추가함으로써 신호 처리 응용에서 더 나은 성능을 보이도록 설계되었다. AE32000C-Lucida 프로세서는 그림. 1.2와 같은 형태를 지니고 있으며, AE32000C-Lucifer와 비교하여 하드웨어적인 부분에 있어서 개선 사항은 다음과 같다.

- AMBA 2.0 AHB/AMBA 3.0 AXI 버스 지원 (선택 가능)
- JTAG을 통한 소프트웨어/하드웨어 디버깅 지원
- 향상된 성능
  - 분기 예측 기법의 채용
  - 8-bit 단위의 SIMD 기능 강화
  - Write-Buffer와 Line-Fill Buffer를 통한 빠른 cache line-replace 지원
  - Cache locking 지원
  - Scratch-PAD Memory 지원
- 저전력 구현
  - 캐쉬 접근 요청 및 캐쉬 SRAM의 동작 기간을 최적화
  - Loop-Buffer 채용으로 인하여, 루프 형태에 따라 명령어 캐쉬 접근 없이 프로세서 수행 가능
  - 향상된 2단 TLB 구조
- 구성 가능한 메모리 크기와 연산/메모리 요소

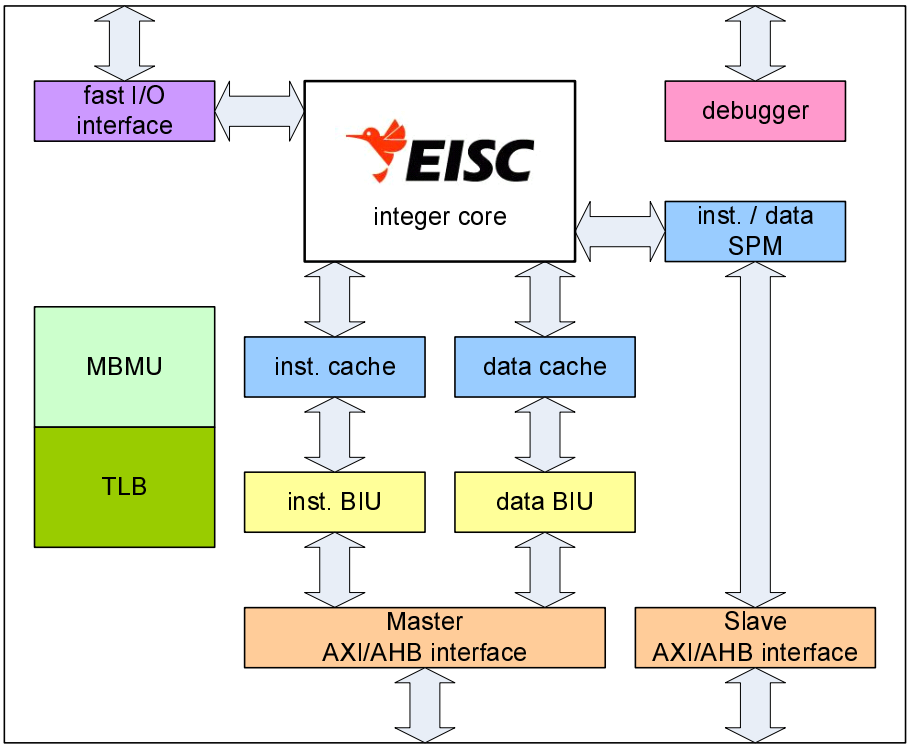


Figure 1.2: Processor Block diagram

1.2.1 기능들

- **EISC Instruction Set: AE32000C**
  AE32000C-Lucida 프로세서는 고성능 32비트 EISC 명령어 셋 아키텍처인 AE32000 중에 SIMD-DSP 기능이 강화된 AE32000C 명령어 셋 아키텍처를 기반으로 하고 있다.

- **DSP Capabilities**
  AE32000C-Lucida 프로세서는 신호처리 응용 분야에서의 성능을 향상시키기 위하여 MAC와 Sum-Of-Produts와 같은 강력한 연산 명령과, 16-bit SIMD와 8-bit SIMD 명령어를 내장하고 있다. 신호 처리 응용에 최적화된 메모리 접근을 지원하기 위한 주소 모드를 제공한다.

- **Memory Management Features**
  AE32000C-Lucida 프로세서는 사용자의 다양한 요구에 부응하기 위하여 펌웨어 수준에서 간단한 메모리 설정과 보호 기능을 수행할 수 있는 MBMU(memory bank management unit)와 더불어 OS를 통하여 세밀한 메모리 설정을 조작할 수 있는 MMU(Memory Management Unit)을 제공하며, MMU의 성능을 강화하기 위하여 2 단계의 TLB(Translation Lookaside Buffer)를 제공한다.

- **Decoupled Instruction Buffer and LERI instruction Folding**
  AE32000C-Lucida 프로세서는 명령어 인출 과정의 메모리 접근 지연에 따른 성능

감소를 줄이고, LERI 명령어 수행을 가속하기 위하여 명령어 인출 단계와 이후의 파이프라인 사이에 명령어 버퍼를 가지고 있다.

- **Dynamic Branch Prediction**
  AE32000C-Lucida 프로세서는 BTB(Branch Target Buffer)를 기반으로 동적 분기 예측을 수행하여 분기에 따른 성능 저하를 최소화 하고 있다.

- **Loop Buffering**
  AE32000C-Lucida 프로세서는 분기 예측 기능을 통하여 일부 Loop를 반복 수행하는 과정에서 추가적인 명령어 메모리에 대한 접근 없이 명령어 버퍼에 대한 접근만으로 프로세서가 수행될 수 있도록 하여 성능 향상을 도모하는 동시에 전력 소모를 줄이고 있다.

- **Configurable Cache**
  AE32000C-Lucida 프로세서는 Soft-IP 형태의 프로세서로써 캐시의 크기와 형태를 조정할 수 있다. AE32000C-Lucida의 캐시는 지정된 메모리 영역에 대하여 각각 캐시의 활성화 여부와 write policy를 조정할 수 있다. 또한 cache locking을 통하여 빈번히 사용되는 부분이 캐시에 지속적으로 존재하게 만들 수 있으며, cache coherency 문제등의 필요에 따라 일부 구간만을 invalidation시킬 수 있다. 또한, 강력한 Line-fill buffer/Write buffer의 채용을 통하여 메모리 접근 지연을 최소화한다.

- **Scratch-PAD Memory**
  AE32000C-Lucida 프로세서는 캐시가 필요치 않은 응용 분야, 혹은 실시간 처리가 강조되는 응용 분야에서 사용할 수 있도록 다양한 형태로 구성된 SPM(Scratch-PAD Memory)을 제공한다. SPM은 프로세서에서 1 cycle에 접근 가능하도록 설정된 메모리이다.

- **AMBA AHB/AXI**
  AE32000C-Lucida 프로세서는 AMBA AHB와 AXI를 지원하므로, 사용자의 요구에 따라 AHB를 기반으로 하는 시스템이나 AXI를 기반으로 하는 시스템 모두에 적용 가능하다.

- **Fast IO interface**
  AE32000C-Lucida 프로세서는 System Bus 이외에 별도의 internal Bus 인터페이스를 가지고 있다. 이를 통하여 Interrupt controller, Timer 등을 Direct로 접근할 수 있다.

- **JTAG debugger**
  JTAG debugger는 EISC프로세서에서 제공하는 ICE(In-Circuit Emulation) 디버깅 기능으로서, 프로그래머에게 프로그램을 디버깅 할 수 있는 기능을 제공한다.

1.3 Tool Support

이 절에서는 AE32000C-Lucida 프로세서를 이용하여 하드웨어/소프트웨어를 개발하고자 할때 사용 가능한 EDA 설계 도구와 Software development tool chain에 대하여 간략히 다루도록 한다.

1.3.1 EDA Tool Support

AE32000C-Lucida 프로세서는 Soft IP형태의 프로세서로서, 특정 EDA 툴에 대한 의존성이 없으므로, 사용자의 편의에 따라 EDA 툴을 선택하여 사용할 수 있다. 단, AE32000C-Lucida 프로세서는 계약의 형태에 따라 RTL 수준 혹은 Protected RTL과 synthesis model이 전달되며, Protected RTL이나 synthesis model이 전달될 때는 사용하는 EDA 툴을 담당 지원 엔지니어에게 통보하여 적합한 모델을 받아야 한다. AE32000C-Lucida는 다음과 같은 EDA 도구에 대하여 환경 및 Script를 지원하고 있으며, 다른 EDA tool에 대해서는 요청에 의하여 구성된다.

- Logic Simulator
  - Cadence®Verilog-XL™,NC-Verilog™, IUS™series
  - Mentor® Modelsim™
- Synthesis and DFT
  - Synopsys® Design Compiler™,DFT Compiler™
  - Synplify® Syplify Pro™
- Static Timing Ananlysis
  - Synsopsys® PrimeTime™
- FPGA prototyping
  - Xilinx® ISE™
  - Altera® Qurtus™

1.3.2 Software Development Support

AE32000C-Lucida 기반의 SoC에서 소프트웨어를 개발할 때 다음과 같은 개발 프로그램이 ADChips에서 지원된다.

- IDE(Integrated Development Environment)
  - EISC Studio™ Develop Suite: EISC 프로세서 기반의 시스템을 개발하기 위한 통합 개발 환경으로서, Microsoft® Windows™ 환경하에서 사용 가능하다. (그림. 1.3 참조)
- GNU 기반의 tool chain: GNU 기반의 컴파일러/디버거/시뮬레이터를 제공한다.

  - GNU C/C++ Compiler
  - GNU macro assembler, linker
  - GNU Debugger (GDB), Insight GUI debugger
  - GNU instruction set simulator

- ESCAsim: EISC system perfromance simulator

개발에 도움을 주는 프로그램 외에도, 개발에 필요한 라이브러리, 예제 코드와 환경에 따른 O/S가 제공되며, 이에 대해서는 담당 엔지니어에게 문의하면 된다.

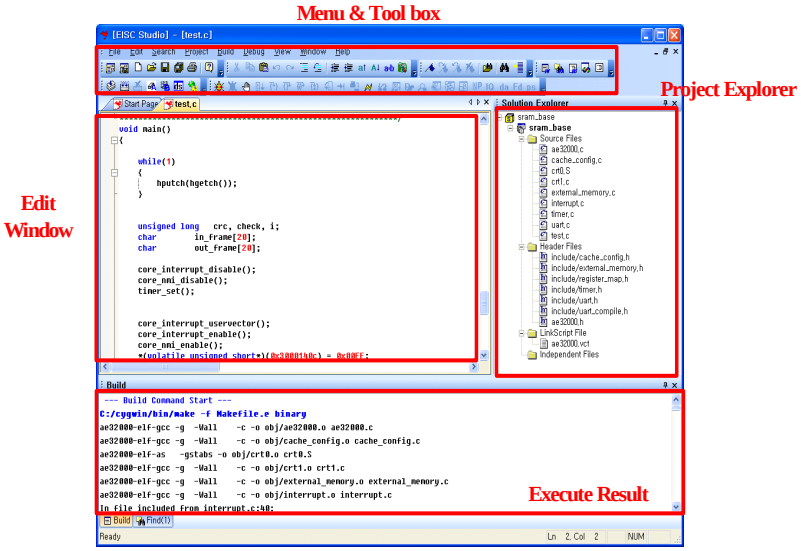


Figure 1.3: EISC Studio,IDE for EISC processor,for Microsoft Windows Environment

1.4 이 문서에 대하여

이 문서는 AE32000C-Lucida 프로세서를 이용하여 하드웨어/소프트웨어를 개발하고자 하는 개발자분들에게 AE32000C-Lucida 프로세서의 정확한 사용 방법을 알려주기 위한 목적으로 개발되었다. 본 문서를 사용함에 있어서 다음과 같은 사항을 참고하라.

- 본 문서는 EISC 명령어 셋 아키텍처 자체나 AE32000 ISA 자체에 대하여 자세히 다루고 있지 않으므로, 해당 사항을 보기 위해서는"AE32000: Instruction Set Architecture Reference Guide"를 참조하십시오.
- 본 문서는 하드웨어 혹은 소프트웨어에서 필요한 상세한 자료를 제공하기보다는 전반적인 구성과 사용법에 대하여 기술하는 문서이므로, 이에 대한 자세한 사항은 "AE32000C-Lucida: Hardware Reference Manual"를 참조하십시오.
- 본 문서는 필요에 의하여 사용자에게 별도의 고지없이 갱신될 수 있습니다.
- 본 문서는 사용자들이 사용하는데 있어서 불편함이 없도록 내용의 정확성을 유지하고 있으나, 하드웨어적인 모델의 갱신이나 문서 기술 상의 오류가 존재할 수 있으며, 본 문서에 기술된 내용 상의 오류로 의한 문제에 대해서는 Advanced Digital Chips Inc.에서 책임지지 않습니다. 따라서, 사용에 있어서 문제가 발생하는 사항은 담당 지원 엔지니어에게 문의/확인하시기 바랍니다.
- 문서의 내용상에 오류를 발견한 경우 eisc@adc.co.kr로 연락주시면 즉시 수정될 수 있도록 조치 하겠습니다.

1.4.1 관련 문서들

- **AE32000C Instruction Set Reference Manual**: AE32000 명령어 셋 아키텍처에 대한 전반적인 사항과 명령어에 대한 사항을 서술한다. AE32000C-Lucida 프로세서 기반의 SoC에서 프로그램을 작성하는 개발자의 경우 필요에 따라 명령어의 자세한 동작에 대하여 이 문서를 통하여 참고할 수 있다.
- **AE32000C-Lucida Hardware Reference Manual**: AE32000C-Lucida의 하드웨어적인 구성 및 설정에 대한 사항을 서술한다. AE32000C-Lucida 프로세서를 이용하여 SoC를 구성하고자 하는 개발자의 경우 필요에 따라 이 문서를 통하여 AE32000C-Lucida의 정확한 동작을 참고할 수 있다.

1.4.2 약어와 용어

1) 숫자의 표시

이 문서에서 대부분의 숫자는 10진수로 표기되어 있다. 단, 다음과 같이 명시적으로 진수를 표기한 경우는 예외로 한다.

- `0x00ff00cc`와 같이 `0x`를 숫자 앞에 같이 기입한 숫자는 16진수를 나타낸다.
- `'h00ff00cc`와 같이 `'h`를 숫자 앞에 같이 기입한 숫자는 16진수를 나타낸다.

- 'b11001010와 같이 'b를 숫자 앞에 같이 기입한 숫자는 2진수를 나타낸다.
- 필요에 따라 'h나 'b 표기법에서는 32'h나 8'b와 같이 '앞에 몇 비트로 표기되는지 명시적으로 표기할 수 있다.

2) 약어**(Acronyms)**

이 문서에서는 문서의 간결성을 위하여 필요한 경우 약어를 이용하도록 한다.

ADChips Advanced Digital Chips Inc.; http://www.adc.co.kr

ASSP Application Specific Standard Product: 특정 응용 분야에 대하여 범용성 있게 사용할 수 있도록 만들어진 SoC 제품.

CISC Complex Instruction Set Computer: 가변 길이 명령어를 이용하여 다양한 동작에 대한 명령어를 정의하여 사용하는 아키텍처. Intel의 x86이 대표적임.

EISC Extendable Instruction Set Computer: ADChips에서 개발된 CISC와 RISC의 특성을 적절히 혼용하여 내장형 응용 분야(embedded application)에 적합하도록 만들어진 아키텍처.

RISC Reduced Instruction Set Computer: 간단한 동작만을 명령어로 구현하고, 이 명령의 처리 효율을 높임으로 전반적인 성능을 향상시킨 아키텍처. HP PA-RISC, Sun-SPARC, IBM Power등이 대표적임.

SoC System-on-a-Chip: 반도체의 고 집적성을 이용하여 전체 시스템을 하나의 Chip 위에 구현하는 것.

EDA Electronic Design Automation

SIMD Single Instruction Multiple Data

MBMU Memory Bank Management Unit: 메모리 뱅크 및 서브 뱅크 단위의 설정을 지원하는 유닛

MMU Memory Management Unit

TLB Translation Lookaside Buffer

1.4.3 표기

이 문서는 사용자의 주의가 필요한 경우, 특정 마크와 색을 이용하여 이를 나타냅니다.

1) 경고

 이 부분에는 경고 문구(Warning Notice)가 들어간다.
이 부분의 경고문은 프로세서의 사용에 있어서 매우 중요한 사항들을 포함하고 있으므로, 반드시 기억하고 있어야 하며, 이 부분의 내용이 무시되는 경우 프로세서가 오동작하거나, 동작/성능에 치명적인 영향을 줄 수 있다.

2) 참고

 이 부분에는 해당 절의 내용을 좀 더 잘 알기 위하여 필요한 참고 노트(reference note)가 들어 있다. 여기서는 아주 간단한 박스 기사(box article) 혹은 반드시 참고하고 넘어가야 할 부분에 대하여 지적하는 부분으로, 필요에 따라 활용하면 된다.

Chapter 2

Programmer's Model

이 장에서 소개하는 Programmer's Model은 일반적으로 Instruction Set Archtiecture(ISA)로도 불린다. 이 장에서는 프로그래머 관점에서의 AE32000C-Lucida 프로세서 기반의 SoC에서 프로그램을 작성하는데 있어서 필요한 사항을 기술하도록 한다. 이 장에서 기술하는 부분 중 명령어에 대한 자세한 사항은 *AE32000C Instuction Set Architecutre Reference Manual*을 참조하면 된다.

2.1 Operation Mode

2.1.1 Operation Mode

프로세서 동작 모드는 사용 자원 관리에 있어서 안정성을 높이기 위하여 사용된다. 프로세서 동작에 영향을 미칠 수 있는 중요한 자원 및 메모리 영역에 접근 권한을 두어, 권한이 없는 사용자 및 프로그램에 의해 프로세서 심각한 위험에 빠지는 것을 막는 것이다. AE32000C-Lucida 프로세서는 다음과 같은 두 가지 모드를 지니고 있다.

- **Supervisor Mode** (관리자 모드)
  관리자 모드는 모든 자원에 대한 접근이 가능한 모드로서, OS와 같이 자원 관리를 담당하는 프로그램들이 사용하는 모드이다.

- **User Mode** (사용자 모드)
  사용자 모드는 사용자 프로그램이 수행되는 모드로서 일반 용도의 프로그램이 구동되는 모드이다. 사용자 모드에서는 사용자에게 할당된 자원에 대한 접근만이 가능하며, 만일 사용자에게 할당되지 않은 자원에 접근하는 경우 access violation이 발생한다. Access violation은 System Coprocessor Exception에 해당하며 자세한 내용은 2.5절을 참조하기 바란다.

각 프로세서 동작 모드에 따라 각각의 스텍 포인터(SP;Stack Pointer)를 지니고 있으며, 나머지 범용 레지스터 및 특수 목적 레지스터는 공유된다. 각 프로세서 동작 모드를 사용하기 위해서는 프로그래밍 초기에 해당 스텍 포인터를 지정해 주어야만 한다. 자세한 내용은 29 page를 참조하기 바란다.

2.1.2 프로세서 동작 모드의 변경

프로세서를 사용하는 과정에서 프로세서의 동작 모드를 변경해야 하는 경우가 발생한다. 프로세서의 동작 모드를 변경하는 가장 빠르고 간단한 방법은 SET/CLR 명령을 이용하는 방법이다. 이 방법은 가장 빠르게 프로세서의 동작 모드를 변경시킬 수 있다는 장점을 지니지만, 파이프라인 프로세서에서 동작에 교란을 줄 수 있다는 위험성을 내포하고 있으므로 반드시 SYNC 명령과 같이 사용되어야 한다. 일반적으로 SET 명령을 통한 프로세서 동작 모드의 변경은 프로그램 실행 초기에 사용자 모드로의 진입을 위하여 사용되는 경우 이외에는 되도록 사용을 자제 해야 한다.

사용자 모드에서 관리자 모드로의 진입은 인터럽트를 이용하여 수행될 수 있다. 다음은 인터럽트를 이용하여 프로세서 동작 모드를 변경하는 예를 나타낸다. SWI Exception에 의해 스텍 영역 저장된 상태 레지스터(Status Register(SR))의 Mode bit[1]를 간접적으로 변경하여 프로세서 동작 모드를 변경한다.

```
1  // process mode is changed to user mode
2  #pragma interrupt_handler
3  void swi0(void)
```

[1] SR의 15,9번 비트를 나타내며, 자세한 내용은 2.2.1절의 SR 부분을 참조하기 바란다.

```
4   {
5       __asm__ __volatile__("lea (%sp,0),  %r6");
6       __asm__ __volatile__("lea (%r6,84), %r0");    //R0 : stack pointer where SR live
7       __asm__ __volatile__("ld  (%r0,0),  %r1");    //R1 : SR
8       __asm__ __volatile__("or  0x8000,   %r1");    // change SR to user mode
9       __asm__ __volatile__("and 0xfffffdff, %r1");  // SR[15]==1, SR[9]==0
10      __asm__ __volatile__("st  %r1,  (%r0,0)");
11  }
12
13  // process mode is changed to supervisor mode
14  #pragma interrupt_handler
15  void swi1(void)
16  {
17      __asm__ __volatile__("lea (%sp,0),  %r6");
18      __asm__ __volatile__("lea (%r6,84), %r0");    //R0 : stack pointer where SR live
19      __asm__ __volatile__("ld  (%r0,0),  %r1");    //R1 : SR
20      __asm__ __volatile__("and 0xffff7fff, %r1");  // change SR to supervisor mode
21      __asm__ __volatile__("and 0xfffffdff, %r1");  // SR[15]==0, SR[9]==0
22      __asm__ __volatile__("st  %r1,  (%r0,0)");
23  }
24
```

2.2 Registers

AE32000C-Lucida 프로세서에서는 9개의 특수 목적 레지스터(SPR;Special Purpose Register), 2개의 스택 포인터(SP;Stack Pointer), 16개의 범용 레지스터(GPR;General Purpose Register)가 제공된다.

- **SPR** : SR, PC, LR, ER, MH, ML, MRE, CR0, CR1
- **SP** : SSP, USP
- **GPR** : R0, R1, R2, ... , R14, R15

2.2.1 Special Purpose Registers

AE32000C-Lucida 프로세서는 다음과 같은 9개의 특수 목적 레지스터와 프로세서 동작 모드에 따른 2개의 스택 포인터를 제공한다.

- **Status Register(SR)**
  AE32000C-Lucida 프로세서의 상태 레지스터는 프로세서에 대한 기본적인 사항과 프로세서의 운용에 대한 정보를 지니고 있다. 그림.2.1과 표.2.1은 상태 레지스터의 각 필드를 나타낸다.

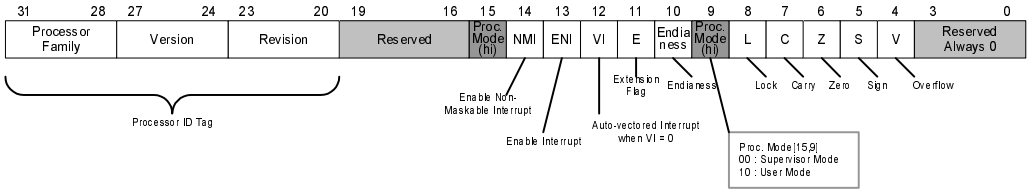


Figure 2.1: 상태 레지스터 정보

Table 2.1: 상태 레지스터의 각 비트 정의 및 설정 방법

Tag	Width	Description
Processor Family	4bit	프로세서의 종류를 나타낸다. AE32000C-Lucida 프로세서의 경우 AE32000 Family로서 '4'번을 지닌다.
Version	4bit	프로세서의 버전을 나타낸다. AE32000C-Lucida 프로세서의 경우 '3' 값을 지닌다.
Revision	4bit	프로세서의 **revision**을 나타낸다. Revision은 동일한 명령어 셋 아키텍쳐 및 구성에서 마이크로 아키텍쳐적으로 작은 수정이 가해진 경우이다. AE32000C-Lucida 프로세서는 '1' 값을 지닌다.

Table 2.1: 상태 레지스터의 각 비트 정의 및 설정 방법(계속)

Tag	Width	Description
Proc Mode	2bit	**Processor Operation Mode**를 의미한다. SR의 15번째 bit와 9번째 bit의 조합으로서 구성되며, 프로세서의 현재 동작을 나타낸다. SET/CLR 명령을 이용하여 프로세서의 상태를 변경시킬 수 있으나, 이는 권장되지 않는다. 자세한 내용은 2.1.2절을 참조하기 바란다. Bit Setting 00 : Supervisor Mode 01 : Reserved 10 : User Mode 11 : Undefined Mode
NMI	1bit	**Non Maskable Interrupt Enable**을 의미한다. Non Maskable Interrupt(NMI)를 받아들일 것인지 결정한다. Bit Setting 0 : Disalbe 1 : Enable
INT	1bit	**Interrupt Enable**을 의미한다. 외부 인터럽트를 받아들일 것인지 결정한다. Bit Setting 0 : Disable 1 : Enable
VI	1bit	**Vectored Interrupt Mode**를 의미한다. 외부 인터럽트의 처리 방식을 vectored 형식으로 할 것인지 auto vectored 형식으로 할 것인지 결정한다. Bit Setting 0 : Autovectored Interrupt 1 : Vectored Interrupt
E	1bit	**Extension Flag**를 의미한다. Bit Setting 0 : Normal Instruction 1 : Extenstion Instruction

Table 2.1: 상태 레지스터의 각 비트 정의 및 설정 방법(계속)

Tag	Width	Description
Endian	1bit	**Endianess**를 의미한다. 프로세서의 Endian 형식을 결정한다. 이 비트는 Power-On Configuration으로서 동작 중간에 변경할 수 없다. Bit Setting 0 : Little Endian Mode 1 : Big Endian Mode
L	1bit	**Lock Flag**를 의미한다. Lock은 프로세서의 Automic processing 동작에 설정된다. 이 bit이 설정되어 있는 경우 프로세서는 NMI와 INT가 disable된 것 처럼 동작한다. Bit Setting 0 : Disable 1 : Enable - Locking Mode
C	1bit	**Carry Flag**를 의미한다. 이전 연산의 결과에서 Carry가 발생한 경우 설정된다. Bit Setting 0 : Carry Not Occured 1 : Carry Occured
Z	1bit	**Zero Flag**를 의미한다. 이전 연산의 결과가 '0'인 경우 설정된다. Bit Setting 0 : Result isn't Zero 1 : Result is Zero
S	1bit	**Sign Flag**를 의미한다. 이전 연산의 결과가 음수인 경우 설정된다. Bit Setting 0 : Positive result 1 : Negative result

Table 2.1: 상태 레지스터의 각 비트 정의 및 설정 방법(계속)

Tag	Width	Description
V	1bit	**Overflow Flag**를 의미한다. 이전 연산의 과정에서 Overflow가 발생한 경우 설정된다. Bit Setting 0 : Overflow Not Occured 1 : Overflow Occured

- **Program Counter(PC) Register**
  프로그램의 수행 위치를 가르키는 레지스터로서 다음에 수행되어야 할 명령의 주소를 지니고 있다. 단, 파이프라인 프로세서에서는 여러 개의 명령이 동시에 수행되고 있으므로, ID 단계에서 수행되는 명령을 기준으로 PC값이 설정되며, 분기가 결정된 경우 분기 목적 주소가 PC로 설정된다.

  주의) 0bit은 항상 '0' 값을 가진다

- **Link Register(LR)**
  Jump and Link 형식의 분기(JAL, JALR명령)를 수행하는 경우 복귀할 주소(link address)를 저장한다. 분기에서의 복귀는 JPLR명령을 이용하여 LR에 저장된 주소로 분기할 수 있으며, 만일 중첩된 분기에서 복귀할 때는 PUSH LR, POP PC의 명령어 Sequence를 이용할 수도 있다.

  주의) 0bit은 항상 '0'값을 가진다

- **Extenstion Register(ER)**
  LERI[2]명령을 통하여 설정되며, 즉치 값이나 오프셋을 확장하는데 이용되며, 이 레지스터의 값은 SR의 E-flag가 '1'인 경우에만 valid하다. AE32000 계열의 프로세서는 LERI Folding기능을 지니고 있으므로, 아키텍쳐/마이크로아키텍쳐 수준에서 실제로 사용하지는 않는다. 따라서, ER을 PUSH하는 경우 '0'의 값이 PUSH되며, POP의 경우에도 프로세서의 동작상에 영향을 주지 않는다. 따라서, 실제적인 아키텍쳐 수준에서 ER은 dummy로 동작한다.

  단, UDI(Undefined Instruction Interrupt)의 경우나 UII(Unimplemented Instruction Interrupt)에서는 해당 명령을 인터럽트 핸들러에서 분석하여 동작을 수행하기 위하여 ER의 값을 GAP(General Access Pointer) 형태로 보존한다.

- **Multiply Register(Multiply High(MH), Multiply Low(ML), Multiply Result Extend(MRE))**
  곱셈/MAC 연산의 결과를 저장하는 레지스터로서 64비트의 크기를 지닌다. 상위 32비트는 MH로, 하위 32비트는 ML로 지정된다. MAC 연산의 경우 이 레지스터들은 accumulation register로서 사용되며, DSP_EXT 옵션을 사용한 경우 accumulator

[2]Load Extension Register with Immediate : 즉치 값으로서 ER을 확장하는 명령

의 정확성을 잃지 않기 위하여 MRE 레지스터가 부가적으로 사용될 수 있다. MRE 레지스터는 일반적인 PUSH/POP에 의하여 저장/복원될 수 없으므로, 값의 보존이 필요한 경우 반드시 범용 레지스터를 경유하여 PUSH/POP을 수행하여야 한다.

- **Counter Register(CR0, CR1)**
  자동 증가 메모리 지정 모드를 지원하기 위한 레지스터로서, 반복적인 for loop 및 DSP 연산을 효과적으로 지원하기 위하여 사용된다. 다음과 같은 주소 모드가 지원되며, 주소 모드에 따라 각 비트의 용도가 변경된다. 그림. 2.2와 표. 2.2는 주소 모드에 따른 Counter Register의 각 필드 정보를 나타낸다.
  - Auto Increment Mode
  - Auto Increment with End-offset Compare Mode
  - Wrap-around Auto Increment Mode
  - Wrap-around Auto Increment with End Check Mode
  - Bit Reverse Mode

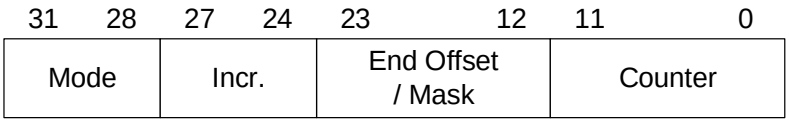


Figure 2.2: Counter Register의 각 필드 정의

Table 2.2: Counter Register의 각 필드 정의 및 동작

Tag	Width	Description
Mode	4bit	**Counter Register**의 동작 모드를 결정한다. 주소 모드를 지정하며, 지정된 모드에 따라 Counter Register의 다른 필드의 역할이 결정된다. Bit Setting 0000 : Auto Increment Mode 0001 : Auto Increment with End-offset Compare Mode 0100 : Wrap-around Auto Increment Mode 0101 : Wrap-around Auto Increment with End Check Mode 0110 : Bit Reverse Mode

Table 2.2: Counter Register의 각 필드 정의 및 동작(계속)

Tag	Width	Description
Incrementor	4bit	**Counter**의 증분을 나타낸다. Counter에 더해질 계수를 지정한다. 현재는 항상 '4'의 값이 사용된다. Bit Setting 0000 : 4
End Offset	12bit	최종 값을 나타낸다. Auto Increment with End-offset Compare Mode에서 최종 값으로 사용된다. Counter 값이 최종 값과 일치하는 경우 Zero flag가 설정된다.
Mask	12bit	몇 비트 연산을 수행할 것인지 결정한다. Wrap-around Mode 및 Bit Reverse Mode에서 몇 비트 연산을 수행할 것인지 결정한다. Bit Reverse Mode에서는 Mask의 크기를 8비트 이하만 지원한다.
Counter	12bit	**Counter**의 값을 나타낸다. 실제적인 카운터의 값을 지니고 있다. 이 값이 index register와 더해져서 메모리 접근 주소로 사용된다.

- **Stack Pointer Register(SSP, USP)**

  AE32000C-Lucida 프로세서는 각 프로세서 동작 모드에 해당하는 스텍 포인터를 지닌다. 스텍 포인터는 스텍 영역의 최상위 위치(가장 최근에 PUSH된 entry의 주소)를 지정한다. 스텍의 성장은 항상 상위 주소에서 하위 주소 방향으로 이루어진다.(PUSH시 주소가 감소한다.)

  스텍 포인터는 프로세서 동작 모드에 따라 shadowing되며, 현재 프로세서 동작 모드 이외의 스텍 포인터는 시스템 보조프로세서의 레지스터(SCPR13)를 통하여 접근이 가능하다.

  - **SCPR13 : User Stack Pointer**

    Supervisor Mode 에서 USP에 접근할 수 있도록 한다.

  각 프로세서 동작 모드를 사용하기 위해서는 프로그래밍 초기에 해당 스텍 포인터를 지정해 주어야만 한다. 이때 LEA명령을 이용해 현재 프로세서 동작 모드에 해당하는 스텍 포인터 값을 설정할 수 있다. 프로그램에서 하나 이상의 프로세서 동작 모드를 사용할 경우, 각 동작 모드를 변경해 가며 LEA 명령을 이용해 각 동작 모드에 해당하는 스텍 포인터 값을 설정할 수 있다. 하지만, 아래의 예와 같이 시스템 보조프로세서 레지스터를 이용하여 동작 모드의 변경 없이 다른 동작 모드의 스텍 포인터를 설정할 수도 있다. 이때 시스템 보조프로세서 레지스터를 접근하기 위한 MVTC 명령이 사용된다.

```
/* initialize stack pointer */

    ldi   _stack-8, %r8
    lea   (%r8),    %sp     /* Supervisor Stack Pointer */

    lea   (%r8,0),  %r0
    ldi   0xf0000,  %r1
    sub   %r1,      %r0
    mvtc  0,        %r13    /* User Stack Pointer */
```

2.2.2 General Purpose Registers

AE32000C-Lucida 프로세서에서는 16개의 범용 레지스터가 제공된다. 범용 레지스터는 아무런 제약 없이 사용할 수 있도록 설계되어 있다. 그러나 C Program과 같이 사용한다면 몇몇 범용 레지스터의 사용에 있어서 주의가 요구된다. 5번 레지스터(R5)는 frame pointer로서 지정되어 있으므로, 인라인 어셈블러에서 이 레지스터를 사요할 때는 주의를 기울여야 한다. 8, 9번 레지스터(R8, R9)는 argument register[3]로 사용되므로, 인자에 대한 처리를 인라인 어셈블러를 통하여 하고자 할 때 이 레지스터들의 사용에 주의하여야 한다. 2개 이상의 인자가 사용될 경우에는 부족한 인수들을 스텍을 이용하여 전달한다. 자세한 내용은 “Software User Guide”를 참조하기 바란다.

[3]argument register의 개수는 Complier 최적화에 따라 추후 변경될 수 있다.

2.3 Instruction Set Highlight

AE32000C-Lucida 프로세서는 고성능 32비트 EISC 명령어 셋 아키텍처인 AE32000 중에 SIMD-DSP 기능이 강화된 AE32000C 명령어 셋 아키텍처를 기반으로 하고 있다. 또한, binary 수준의 하위 호환성을 지니고 있다. 본 절에서는 명령어의 binary encoding과 간략한 명령어의 동작에 대하여 살펴보도록 한다. 명령어에 대한 자세한 동작 및 주의사항은 "AE32000C Instruction Set Architecture Reference Manual"을 참조하기 바란다.

2.3.1 Binary Encoding and Mnemonic

AE32000C-Lucida 프로세서의 binary encoding과 간략한 Mnemonic은 표. 2.3과 같다. Mnemonic에 사용된 지시어들은 다음과 같은 뜻을 가진다.

- **dst**

  목적 레지스터(destination register)로 범용 레지스터 중의 하나를 뜻한다. 연산의 결과를 저장하는 레지스터이다.

- **src, src1, src2**

  원천 레지스터(source register)로 범용 레지스터 중의 하나를 뜻한다. 연산에 사용되는 오퍼랜드를 의미한다.

- **idx**

  인덱스 레지스터(index register)로 범용 레지스터 중의 하나를 뜻한다.[4] index addressing에서 인덱스 값으로 사용된다.

- **offset**

  즉치 값(immediate value)으로 정수 값을 가진다. index addressing, stack addressing, pc reltive program memory addressing에서 변위(offset) 값으로 사용된다.

- **imm**

  즉치 값(immediate value)으로 정수 값을 가진다. 연산에 직접 사용되거나 LERI 명령에 의해 확장되어 연산에 사용된다.

- **list**

  PUSH, POP 명령에 사용되는 범용 레지스터, 특수 목적 레지스터의 집합을 나타낸다.

- **sftamt**

  쉬프트 연산에 사용되는 쉬프트 횟수를 나타낸다. 정수 값으로 0에서 31 사이의 값을 가진다.

[4] LDAU, STAU 명령에서는 다른 의미를 가진다.

- **sftreg**

  쉬프트 연산에 사용되는 쉬프트 횟수를 간직하는 레지스터로 범용 레지스터 중의 하나를 뜻한다. 32비트 레지스터 가운데 하위 5비트가 사용된다.

- **flag pos**

  SET, CLR 명령에 사용되는 정수 값으로 상태 레지스터의 0에서 15 사이의 각 비트를 나타낸다.

- **or**

  Counter Register의 number를 나타낸다.

- **cpno**

  보조프로세서(Coprocessor)의 number를 나타낸다. 0에서 3 사이의 정수 값을 가진다.

- **src_grp**

  UNPACK 연산에 사용되는 연속된 두개의 범용 레지스터를 뜻한다. 하위에 0과 1이 생략된 두 개의 범용 레지스터를 나타낸다.(예. 010 - 4,5번 레지스터)

표. 2.3에서 음영으로 표시된 부분은 AE32000C-Lucida 프로세서에서 DSP_EXT 옵션을 사용한 경우 사용가능한 명령으로 사용시 주의가 필요하다.

Table 2.3: AE32000C-Lucida 프로세서 Instructions : binary encoding

Instruction	f	e	d	c	b	a	9	8	7	6	5	4	3	2	1	0	Mnemonic
LD	0	0	0	0	dst				offset				idx				LD (idx, off), dst
ST	0	0	0	1	src				offset				idx				ST src, (idx, off)
LDB	0	0	1	0	dst				0	offset			idx				LDB (idx, off), dst
LDS	0	0	1	0	dst				1	offset			idx				LDS (idx, off), dst
STB	0	0	1	1	src				0	offset			idx				STB src, (idx, off)
STS	0	0	1	1	src				1	offset			idx				STS src, (idx, off)
LERI	0	1	imm														LERI imm
LDBU	1	0	0	0	dst				0	offset			idx				LDBU (idx, off), dst
LDSU	1	0	0	0	dst				1	offset			idx				LDSU (idx, off), dst
LD	1	0	0	1	dst				0	offset							LD (SP, off), dst
ST	1	0	0	1	src				1	offset							ST src, (SP, off)
LDI	1	0	1	0	dst				imm								LDI imm, dst
PUSH	1	0	1	1	0	0	0	0	REG LIST (7,6,5,4,3,2,1,0)								PUSH list
PUSH	1	0	1	1	0	0	1	0	(15,14,13,12,11,10,9,8)								PUSH list
PUSH	1	0	1	1	0	1	0	0	(SR,PC,LR,ER,MH,ML,CR1,CR0)								PUSH list
POP	1	0	1	1	0	0	0	1	REG LIST (7,6,5,4,3,2,1,0)								POP list
POP	1	0	1	1	0	0	1	1	(15,14,13,12,11,10,9,8)								POP list
POP	1	0	1	1	0	1	0	1	(SR,PC,LR,ER,MH,ML,CR1,CR0)								POP list

Table 2.3: AE32000C-Lucida 프로세서 Instructions : binary encoding(계속)

Instruction	f	e	d	c	b	a	9	8	7	6	5	4	3	2	1	0	Mnemonic
LEA(ASPO)	1	0	1	1	0	1	1	0	offset								LEA (SP, imm), SP
MRS	1	0	1	1	0	1	1	1	0	imm			dst				MRS imm, dst
	1	0	1	1	0	1	1	1	1								Reserved
ADD	1	0	1	1	1	0	0	0	dst				src/imm				ADD src/imm, dst
ADC	1	0	1	1	1	0	0	1	dst				src/imm				ADC src/imm, dst
SUB	1	0	1	1	1	0	1	0	dst				src/imm				SUB src/imm, dst
SBC	1	0	1	1	1	0	1	1	dst				src/imm				SBC src/imm, dst
AND	1	0	1	1	1	1	0	0	dst				src/imm				AND src/imm, dst
OR	1	0	1	1	1	1	0	1	dst				src/imm				OR src/imm, dst
XOR	1	0	1	1	1	1	1	0	dst				src/imm				XOR src/imm, dst
CMP	1	0	1	1	1	1	1	1	dst				src/imm				CMP src/imm, dst
ASR	1	1	0	0	dst				0	sftamt					0	0	ASR sftamt, dst
LSR	1	1	0	0	dst				0	sftamt					0	1	LSR sftamt, dst
ASL	1	1	0	0	dst				0	sftamt					1	0	ASL sftamt, dst
SSL	1	1	0	0	dst				0	sftamt					1	1	SSL sftamt, dst
ASR	1	1	0	0	dst				1	0	sftreg				0	0	ASR sftreg, dst
LSR	1	1	0	0	dst				1	0	sftreg				0	1	LSR sftreg, dst
ASL	1	1	0	0	dst				1	0	sftreg				1	0	ASL sftreg, dst
SSL	1	1	0	0	dst				1	0	sftreg				1	1	SSL sftreg, dst
ADDQ	1	1	0	0	dst				1	1	0	imm					ADDQ imm, dst
CMPQ	1	1	0	0	src				1	1	1	imm					CMPQ imm, src
JNV	1	1	0	1	0	0	0	0	offset								JNV label
JV	1	1	0	1	0	0	0	1	offset								JV label
JP	1	1	0	1	0	0	1	0	offset								JP label
JM	1	1	0	1	0	0	1	1	offset								JM label
JNZ	1	1	0	1	0	1	0	0	offset								JNZ label
JZ	1	1	0	1	0	1	0	1	offset								JZ label
JNC	1	1	0	1	0	1	1	0	offset								JNC label
JC	1	1	0	1	0	1	1	1	offset								JC label
JGT	1	1	0	1	1	0	0	0	offset								JGT label
JLT	1	1	0	1	1	0	0	1	offset								JLT label
JGE	1	1	0	1	1	0	1	0	offset								JGE label
JLE	1	1	0	1	1	0	1	1	offset								JLE label
JHI	1	1	0	1	1	1	0	0	offset								JHI label
JLS	1	1	0	1	1	1	0	1	offset								JLS label
JMP	1	1	0	1	1	1	1	0	offset								JMP label
JAL	1	1	0	1	1	1	1	1	offset								JAL label
EXTB	1	1	1	0	0	0	0	0	0	0	0	0	dst				EXTB dst
EXTS	1	1	1	0	0	0	0	0	0	0	0	1	dst				EXTS dst
MFMRE	1	1	1	0	0	0	0	0	0	0	1	0	dst				MFMRE dst
MTMRE	1	1	1	0	0	0	0	0	0	0	1	1	src				MTMRE src
CVB	1	1	1	0	0	0	0	0	0	1	0	0	dst				CVB dst
CVS	1	1	1	0	0	0	0	0	0	1	0	1	dst				CVS dst
NEG	1	1	1	0	0	0	0	0	0	1	1	0	dst				NEG dst
NOT	1	1	1	0	0	0	0	0	0	1	1	1	dst				NOT dst
JR	1	1	1	0	0	0	0	0	1	0	0	0	src				JR src

Table 2.3: AE32000C-Lucida 프로세서 Instructions : binary encoding(계속)

Instruction	f	e	d	c	b	a	9	8	7	6	5	4	3	2	1	0	Mnemonic
JALR	1	1	1	0	0	0	0	0	1	0	0	1	src				JALR src
JPLR	1	1	1	0	0	0	0	0	1	0	1	0					JPLR
NOP	1	1	1	0	0	0	0	0	1	0	1	1					NOP
SWI	1	1	1	0	0	0	0	0	1	1	0	0	imm				SWI exception_no
STEP	1	1	1	0	0	0	0	0	1	1	0	1					STEP
HALT	1	1	1	0	0	0	0	0	1	1	1	0	imm				HALT halt_level
BRKPT	1	1	1	0	0	0	0	0	1	1	1	1					BRKPT
CNT0	1	1	1	0	0	0	0	1	0	0	0	0	src				CNT0 src
CNT1	1	1	1	0	0	0	0	1	0	0	0	1	src				CNT1 src
SET	1	1	1	0	0	0	0	1	0	0	1	0	flag pos				SET position
CLR	1	1	1	0	0	0	0	1	0	0	1	1	flag pos				CLR position
MTML	1	1	1	0	0	0	0	1	0	1	0	0	src				MTML src
MTMH	1	1	1	0	0	0	0	1	0	1	0	1	src				MTMH src
MFML	1	1	1	0	0	0	0	1	0	1	1	0	dst				MFML dst
MFMH	1	1	1	0	0	0	0	1	0	1	1	1	dst				MFMH dst
MTCR0	1	1	1	0	0	0	0	1	1	0	0	0	src				MTCR0 src
MTCR1	1	1	1	0	0	0	0	1	1	0	0	1	src				MTCR1 src
SYNC	1	1	1	0	0	0	0	1	1	0	1	0					SYNC
	1	1	1	0	0	0	0	1	1	0	1	1					Reserved
LSEA	1	1	1	0	0	0	0	1	1	1	0	0	idx				LSEA (idx,imm), SP
LESA	1	1	1	0	0	0	0	1	1	1	0	1	dst				LESA (SP, imm), dst
MFCR0	1	1	1	0	0	0	0	1	1	1	1	0	dst				MFCR0 dst
MFCR1	1	1	1	0	0	0	0	1	1	1	1	1	dst				MFCR1 dst
AVGB	1	1	1	0	0	0	1	0	dst			0	src/imm				AVGB src/imm, dst
AVGS	1	1	1	0	0	0	1	0	dst			1	src/imm				AVGS src/imm, dst
TST	1	1	1	0	0	0	1	1	src2				src1				TST src1, src2
LEA	1	1	1	0	0	1	0	0	dst				idx				LEA (idx, imm), dst
ROR	1	1	1	0	0	1	0	1	0	imm			dst				ROR imm, dst
ROL	1	1	1	0	0	1	0	1	1	imm			dst				ROL imm, dst
MUL	1	1	1	0	0	1	1	0	dst			0	src/imm				MUL src/imm, dst
MULU	1	1	1	0	0	1	1	0	dst			1	src/imm				MULU src/imm, dst
MAC	1	1	1	0	0	1	1	1	src2				src1				MAC src1, src2
LDB	1	1	1	0	1	0	0	0	0	offset			dst				LDB (SP,imm), dst
LDS	1	1	1	0	1	0	0	0	1	offset			dst				LDS (SP,imm), dst
LDBU	1	1	1	0	1	0	0	1	0	offset			dst				LDBU (SP,imm), dst
LDSU	1	1	1	0	1	0	0	1	1	offset			dst				LDSU (SP,imm), dst
STB	1	1	1	0	1	0	1	0	0	offset			src				STB src, (SP,imm)
STS	1	1	1	0	1	0	1	0	1	offset			src				STS src, (SP,imm)
LDAU	1	1	1	0	1	0	1	1	0	idx		or	dst				LDAU CRNO, idx, dst
STAU	1	1	1	0	1	0	1	1	1	idx		or	src				STAU CRNO, src, idx
CPCMD	1	1	1	0	1	1	cpno		0	0	cpcmd						CPCMD CPNO, cpcmd
GETCn	1	1	1	0	1	1	cpno		0	1	0	0	status				GETC CPNO, bit_position
EXECn	1	1	1	0	1	1	cpno		0	1	0	1	status				EXEC CPNO, bit_position
MVTCn	1	1	1	0	1	1	cpno		0	1	1	0	dst				MVTC CPNO, cpdst
MVFCn	1	1	1	0	1	1	cpno		0	1	1	1	src				MVFC CPNO, cpsrc
LDCn	1	1	1	0	1	1	cpno		1	0	0	0	dst				LDC CPNO, (SP,imm), cpdst

Table 2.3: AE32000C-Lucida 프로세서 Instructions : binary encoding(계속)

Instruction	f	e	d	c	b	a	9	8	7	6	5	4	3	2	1	0	Mnemonic
LDCn	1	1	1	0	1	1	cpno		1	0	0	1	dst				LDC CPNO, (idx,imm), cpdst
STCn	1	1	1	0	1	1	cpno		1	0	1	0	src				STC CPNO, cpsrc, (SP,imm)
STCn	1	1	1	0	1	1	cpno		1	0	1	1	src				STC CPNO, cpsrc, (idx,imm)
	1	1	1	0	1	1			1	1							Reserved
MACB	1	1	1	1	0	0	0	0	src2				src1/imm				MACB src1/imm, src2
MACS	1	1	1	1	0	0	0	1	src2				src1/imm				MACS src1/imm, src2
MSOPB	1	1	1	1	0	0	1	0	src2				src1/imm				MSOPB src1/imm, src2
MSOPS	1	1	1	1	0	0	1	1	src2				src1/imm				MSOPS src1/imm, src2
SADDB	1	1	1	1	0	1	0	0	dst				src/imm				SADDB src/imm, dst
SADDS	1	1	1	1	0	1	0	1	dst				src/imm				SADDS src/imm, dst
SADUB	1	1	1	1	0	1	1	0	dst				src/imm				SADUB src/imm, dst
SADUS	1	1	1	1	0	1	1	1	dst				src/imm				SADUS src/imm, dst
SADD	1	1	1	1	1	0	0	0	dst				src/imm				SADD src/imm, dst
UPKLS	1	1	1	1	1	0	0	1	src_grp			0	dst				UPKLS src_grp, dst
UPKHS	1	1	1	1	1	0	0	1	src_grp			1	dst				UPKHS src_grp, dst
MIN	1	1	1	1	1	0	1	0	dst			0	src/imm				MIN src/imm, dst
MAX	1	1	1	1	1	0	1	0	dst			1	src/imm				MAX src/imm, dst
ABS	1	1	1	1	1	0	1	1	0	0	0	0	dst				ABS dst
	1	1	1	1	1	0	1	1									Reserved
UPK0LB	1	1	1	1	1	1	0	0	src_grp			0	dst				UPK0LB src_grp, dst
UPK0HB	1	1	1	1	1	1	0	0	src_grp			1	dst				UPK0HB src_grp, dst
UPK1LB	1	1	1	1	1	1	0	1	src_grp			0	dst				UPK1LB src_grp, dst
UPK1HB	1	1	1	1	1	1	0	1	src_grp			1	dst				UPK1HB src_grp, dst
UPK2LB	1	1	1	1	1	1	1	0	src_grp			0	dst				UPK2LB src_grp, dst
UPK2HB	1	1	1	1	1	1	1	0	src_grp			1	dst				UPK2HB src_grp, dst
UPK3LB	1	1	1	1	1	1	1	1	src_grp			0	dst				UPK3LB src_grp, dst
UPK3HB	1	1	1	1	1	1	1	1	src_grp			1	dst				UPK3HB src_grp, dst

2.3.2 Memory Access Instructions

AE32000C-Lucida 프로세서에서 지원되는 메모리 접근 명령들은 표. 2.4에 나열되어 있다.

Table 2.4: Memory Access Instructions

Class	Details	Mnemonic
Load	Word	LD (%Ri/%SP, imm), %Rd
	Byte (unsigned)	LDBU (%Ri/%SP, imm), %Rd
	Byte (signed)	LDB (%Ri/%SP, imm), %Rd
	Short (unsigned)	LDSU (%Ri/%SP, imm), %Rd
	Short (signed)	LDS (%Ri/%SP, imm), %Rd
	Word (auto increment)	LDAU CRNO, %Ri, %Rd
Load Multiple	Pop	POP reg list
Store	Word	ST %Rs, (%Ri/%SP, imm)
	Byte	STB %Rs, (%Ri/%SP, imm)

Table 2.4: Memory Access Instructions(계속)

Class	Details	Mnemonic
	Short	STS %Rs, (%Ri/%SP, imm)
	Word (auto increment)	STAU CRNO, %Rs, %Ri
Store Multiple	Push	PUSH reg list

- Byte, Short(Half Word), Word 크기로 접근이 가능하며, Load의 경우 Byte/Short 크기의 데이터를 signed/unsigned extension을 통해 32비트 레지스터에 채울 수 있다.
- PUSH/POP 명령은 하나의 명령을 통해 다수의 레지스터에 대한 Load/Store를 할 수 있다.

2.3.3 Move

AE32000C-Lucida에서 지원되는 레지스터간의 데이터 이동 명령들은 표. 2.5에 나열되어 있다.

Table 2.5: Move Instructions

Class	Details	Mnemonic
Move	Move	MOV %s, %d
	Move with add	LEA (%Rs/%SP, imm), %Rd/%SP
	Move from GPR to MH	MTMH %Rs
	Move from GPR to ML	MTML %Rs
	Move from MH to GPR	MFMH %Rd
	Move from ML to GPR	MFML %Rd
	Move from GPR to MRE	MTMRE %Rs
	Move from MRE to GPR	MFMRE %Rd
	Move from GPR to CR0	MTCR0 %Rs
	Move from GPR to CR1	MTCR1 %Rs
	Move from CR0 to GPR	MFCR0 %Rd
	Move from CR1 to GPR	MFCR1 %Rd

- 범용 레지스터간의 데이터 이동은 MOV 명령을 사용한다.
- 범용 레지스터와 스택 포이터간의 데이터 이동은 LEA 명령을 사용하며, 이동시에 즉치값을 같이 더하여 이동할 수 있다.
- 이때 즉치값은 32비트 singed number 이므로, 실제적으로 가/감산이 가능하다.
- 범용 레지스터와 특수 목적 레지스터간의 데이터 이동은 MTxx, MFxx 명령을 사용하며, 즉치값이 더해질 수 없다.

- 범용 레지스터와 보조 프로세서 레지스터간의 데이터 이동은 2.3.6절에서 다루도록 한다.

2.3.4 Branch

분기 명령은 프로그램의 흐름을 제어하는 동작을 수행하는 명령으로서, 조건 분기와 무조건 분기로 나뉜다. AE32000C-Lucida 프로세서에서는 코드 밀도를 향상시키기 위하여 다양한 형태의 조건 분기를 지원한다. 분기 명령들은 표. 2.6에 나열되어 있다.

Table 2.6: Branch Instructions

Class	Details	Mnemonic
Branch	Jump	JMP label
	Jump and Link	JAL label
	Jump on overflow clear	JNV label
	Jump on overflow set	JV label
	Jump on sign clear (positive or zero)	JP label
	Jump on sign set (negative)	JN label
	Jump on non-zero (not equal)	JNZ label
	Jump on zero (equal)	JZ label
	Jump carry clear (unsigned higher or equal)	JNC label
	Jump carry set (unsigned lower)	JC label
	Jump signed greater	JGT label
	Jump signed less	JLT label
	Jump signed greater or equal	JGE label
	Jump signed less or equal	JLE label
	Jump unsigned higher	JHI label
	Jump unsigned lower or equal	JLS label
	Jump register indirect	JR %Rs
	Jump register indirect and Link	JALR %Rs
	Jump to LR(Link Register)	JPLR

- 분기의 목적 주소는 다음 두 가지 형태로 지정할 수 있다.
  - PC-relative : 현재 PC값을 기준으로 offset 만큼 분기하는 방식으로 가장 일반적으로 사용되는 형태이다. offset의 값은 32bit singend값이 될 수 있어 가/감산이 가능하다.
  - Register indirect : 범용 레지스터의 값을 이용해 분기하는 방식으로서, dynamic linked library 등의 함수를 연결할 때 사용된다. AE32000C-Lucida 프로세서의 구현에서는 PC-relative 방식보다 더 많은 분기 패널티가 존재한다.
- JAL, JALR 명령은 분기의 시점에서 현재 PC값을 LR(Link Register)에 저장하며, JPLR 명령을 통해 복귀할 수 있다.

2.3.5 Arithmetic & Logical

AE32000C-Lucida 프로세서에서는 기본적인 산술 연산 이외에 DSP에서 사용할 수 있는 다양한 연산 기능(32비트 배럴 쉬프트, MAC연산, leading 0/1 count)을 제공한다. 추가적인 DSP 연산은 2.4절에서 다루도록 한다.

Table 2.7: Arithmetic & Logical Instructions

Class	Details	Mnemonic	flag
Arithmetic	Add	ADD %Rs/imm, %Rd	C Z S V
	Add with short immediate	ADDQ imm5, %Rd	C Z S V
	Add with carry	ADC %Rs/imm, %Rd	C Z S V
	Substract	SUB %Rs/imm, %Rd	C Z S V
	Substract with carry	SBC %Rs/imm, %Rd	C Z S V
Logical	AND	AND %Rs/imm, %Rd	Z S
	OR	OR %Rs/imm, %Rd	Z S
	XOR	XOR %Rs/imm, %Rd	Z S
	NOT	NOT %Rd	Z S
	Test	TST %Rs1/imm, %Rs2	Z S
Compare	Compare	CMP %Rs1/imm, %Rs2	C Z S V
	Compare with short immediate	CMPQ imm5, %Rs	C Z S V
Shift	Arithmetic shift right	ASR %Rc/imm5, %Rd	C Z S
	Logical shift right	LSR %Rc/imm5, %Rd	C Z S
	Arithmetic shift left	ASL %Rc/imm5, %Rd	C Z S
	Set shift left	SSL %Rc/imm5, %Rd	C Z S
Multiply	Multiply	MUL %Rs/imm, %Rd	
	Multiply unsigned	MULU %Rs/imm, %Rd	
	Multiply and accumulate	MAC %Rs/imm, %Rs2	
Misc	Extension from byte to word	EXTB %Rd	Z S
	Extension from short to word	EXTS %Rd	Z S
	Convert from word to byte	CVB %Rd	Z S
	Convert from word to short	CVS %Rd	Z S
	Count leading zero	CNT0 %Rs	Z
	Count leading one	CNT1 %Rs	Z

- 각 연산의 결과에 의해 변경되는 flag는 표. 2.7에 표시되어 있다.
- 대부분의 연산 기능의 경우 %Rs 대신 즉치값을 이용하는 것이 가능하며, 이때 즉치값은 32bit singed number가 된다.
- MUL, MULU 연산의 %Rd는 짝수 범용 레지스터(R0, R2, R4,...)만 가능하다.

2.3.6 Coprocessor Access

AE32000C-Lucida 프로세서는 시스템 보조 프로세서 이외에 3개 까지의 보조 프로세서를 연결하는 것이 가능하다. 보조 프로세서를 접근하기 위한 명령들은 표. 2.8에 나열되어 있다.

Table 2.8: Coprocessor Access Instructions

Class	Details	Mnemonic	flag
Coprocessor	Instruction	CPCMD coproc_command	
	Move from Coproc. to GPR	MVFC %Rd@CP	
	Move from GPR to Coproc.	MVTC %Rd@CP	
	Check status bit in Coproc.	GETC imm4	Z
	Load from memory to Coproc.	LDC CPNO, (%Ri, imm), %Rd@CP	
	Store from Coproc. to memory	STC CPNO, %Rs@CP, (%Ri, imm)	
	Exception on Coproc. status	EXEC imm4	Z

- MVTC, MVFC 명령은 범용 레지스터 0번을 이용한 데이터 전달만 가능하다.
- LDC, STC 명령은 시스템 보조 프로세서와 같이 사용될 수 없다.
- LDC, STC 명령의 인덱스 레지스터는 스텍 포인터 또는 범용 레지스터 1번이 이용된다.
- EXEC 명령은 보조 프로세서의 상태 비트를 이용하여 Coprocessor Interrupt를 발생시킨다.

2.4 DSP Acceleration

AE32000C-Lucida 프로세서는 신호처리 응용 분야에서의 성능을 향상시키기 위한 DSP 명령어를 지원한다. 아래에 소개될 DSP 명령어는 **??**절에서 소개된 DSP_EXT 옵션을 사용한 경우 사용가능한 명령어이다.

2.4.1 Multiply and Accumulation

Multiply and Accumulation(MAC) 연산은 곱셈의 결과를 누적하여 더해 나가는 연산으로, 신호처리 응용 분야에서 널리 사용되는 연산이다. AE32000C-Lucida 프로세서에서 지원되는 MAC 연산은 표. 2.9에 나열되어 있다.

Table 2.9: Multiply and Accumulation Instructions

Class	Details	Mnemonic
MAC	Word	MAC (%Rs1/imm), %Rs2
	Short SIMD	MACS (%Rs1/imm), %Rs2
	Byte SIMD	MACB (%Rs1/imm), %Rs2
	Multiple Sum of Product (Short)	MSOPS (%Rs1/imm), %Rs2
	Multiple Sum of Product (Byte)	MSOPB (%Rs1/imm), %Rs2

- MAC Word 연산은 2.3.5절에서 소개된 MAC 연산과 마이크로 아키텍쳐적으로 차이를 보이지만 사용자 입장에서는 동일하게 사용 가능하다.
- SIMD MAC 연산에는 Accumulation 과정에서 정확성을 잃지 않기 위하여 Multiply Result Extend(MRE) 레지스터가 추가적으로 사용된다.
- SIMD MAC 연산의 동작은 그림.2.3에 나타나 있다.
- MSOPB, MSOPS 연산은 SIMD 형식으로 연산된 결과들을 하나로 합치는 역할을 수행하며, 각각 48비트(MH[15:0],ML[31:0]), 32비트(ML[31:0])의 결과를 가진다.

2.4.2 Saturate Arithmetic

Saturate 연산은 그림. 2.4와 같이 일반적인 wrap-around 연산과 다르게 최대값/최소값을 초과한 경우 표현 가능한 최대/최소값을 유지하도록 되어 있는 연산이다. AE32000C-Lucida 프로세서에서 지원되는 saturate 연산은 표. 2.10에 나열되어 있다.

Table 2.10: Saturate Instructions

Class	Details	Mnemonic
Saturate Add	Word	SADD (%Rs/imm), %Rd
	Short (signed)	SADDS (%Rs/imm), %Rd
	Short (unsigned)	SADUS (%Rs/imm), %Rd
	Byte (signed)	SADDB (%Rs/imm), %Rd

Table 2.10: Saturate Instructions(계속)

Class	Details	Mnemonic
	Byte (unsigned)	SADUB (%Rs/imm), %Rd

2.4.3 Unpack

Unpack은 SIMD 구조에서 필요한 연산의 형태를 정렬하기 위하여 사용되는 명령으로서, 필요에 따라 Word 전체에서 사용되는 필드들을 재정렬하여 이후에 사용되는 SIMD 연산의 효율을 높이기 위하여 사용된다. AE32000C-Lucida 프로세서는 Short(16bit) 데이터 타입과 Byte(8bit) 데이터 타입에 대한 SIMD 연산을 지원하므로, 이 두가지 SIMD 형식에 대한 unpack을 지원한다. Unpack 연산은 표. 2.11에 나열되어 있으며, unpack 과정에서 사용되는 연산의 동작은 그림. 2.5에 나타내고 있다.

Table 2.11: Unpack Instructions

Class	Details	Mnemonic
Unpack	Short to high	UPKHS %Rsrc_grp, %Rd
	Short to low	UPKLS %Rsrc_grp, %Rd
	Byte from 0 to high	UPK0HB %Rsrc_grp, %Rd
	Byte from 0 to low	UPK0LB %Rsrc_grp, %Rd
	Byte from 1 to high	UPK1HB %Rsrc_grp, %Rd
	Byte from 1 to low	UPK1LB %Rsrc_grp, %Rd
	Byte from 2 to high	UPK2HB %Rsrc_grp, %Rd
	Byte from 2 to low	UPK2LB %Rsrc_grp, %Rd
	Byte from 3 to high	UPK3HB %Rsrc_grp, %Rd
	Byte from 3 to low	UPK3LB %Rsrc_grp, %Rd

- %Rsrc_grp은 짝수 범용 레지스터(R0, R2, R4,...)만 지정할 수 있으며, 연속된 두개의 레지스터가 사용된다.(R0-R1, R2-R3, ...)
- Byte unpack 명령은 목적 레지스터의 상위 16비트 또는 하위 16비트를 채우는 명령이므로, 하나의 Word를 구성하기 위해서는 2개의 unpack 명령이 사용되어야 한다.

2.4.4 Miscellianeous

AE32000C-Lucida 프로세서는 위에 기술된 DSP 연산 이외에 다수의 DSP 명령을 내장하고 있으며, 표. 2.12에 나열되어 있다.

Table 2.12: DSP 기타 Instructions

Class	Details	Mnemonic
Average	Average SIMD Short	AVGS (%Rs/imm), %Rd

Table 2.12: DSP 기타 Instructions(계속)

Class	Details	Mnemonic
	Average SIMD Byte	AVGB (%Rs/imm), %Rd
Rotate	Rotate Left Rotate Right	ROL imm, %Rd ROR imm, %Rd
MIN/MAX	Minimum Maximum	MIN (%Rs/imm), %Rd MAX (%Rs/imm), %Rd
Absolute Value	Absolute Value	ABS %Rd
Radix Point adjust	Fixed Point Multiply Result Shift	MRS imm, %Rd

- AVGS, AVGB, MIN, MAX 명령은 목적 레지스터를 0에서 7번 사이의 범용 레지스터만 사용 가능하다.

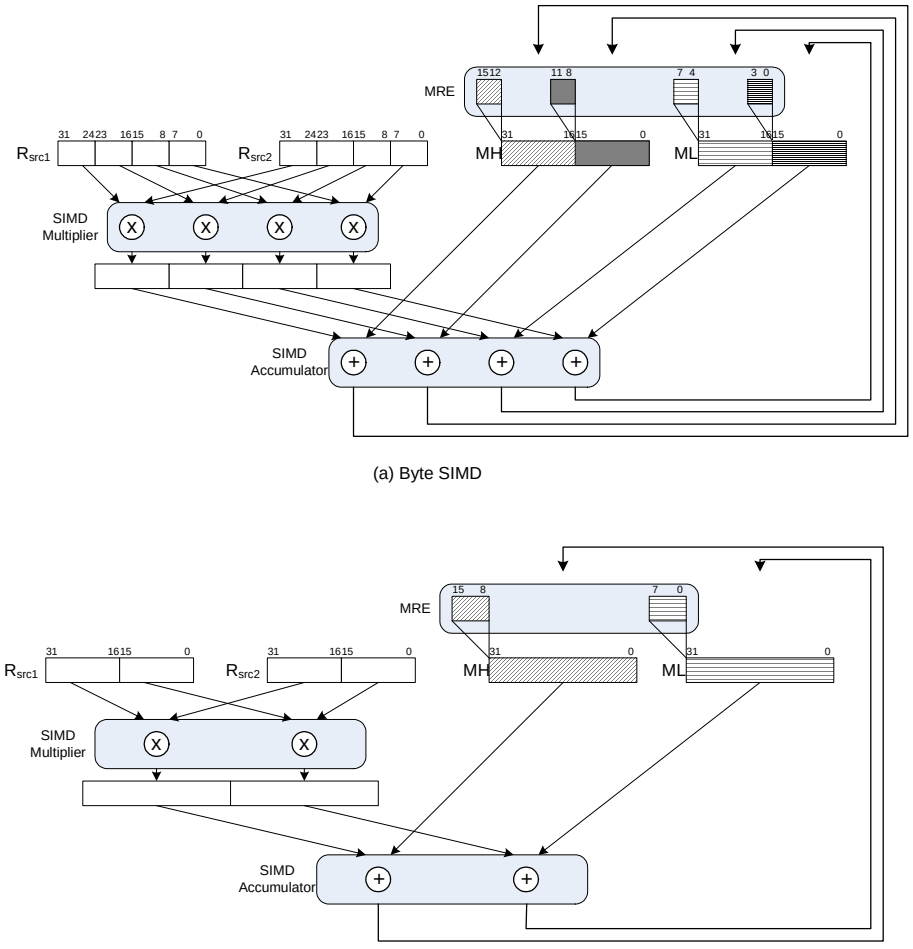


Figure 2.3: SIMD MAC 연산의 동작

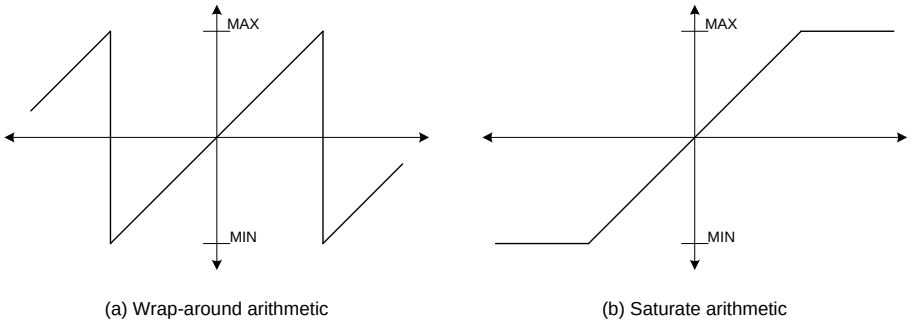
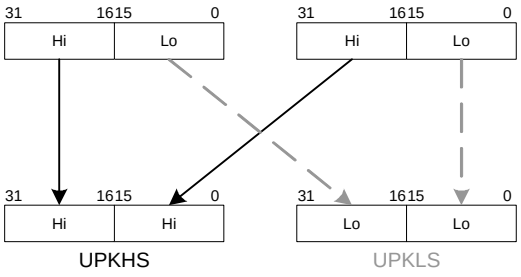
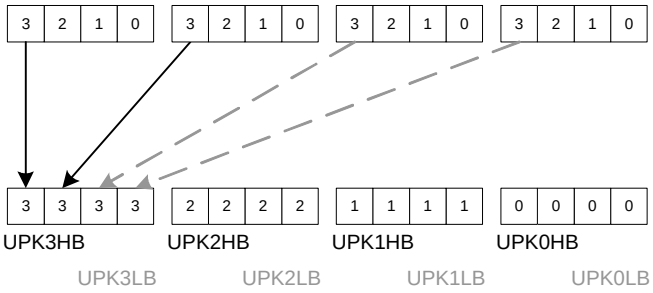


Figure 2.4: Saturate Arithmetic



(a) Unpack (Short data type)



(b) Unpack (Byte data type)

Figure 2.5: Unpack 연산의 동작

2.5 Exceptions

Exception은 프로세서의 정상적인 흐름을 방해하는 모든 상황을 의미한다. AE32000C-Lucida 프로세서는 프로그램 수행중에 exception이 발생하였을 경우, 그림. 2.6과 같은 과정을 통해 exception을 처리한다.

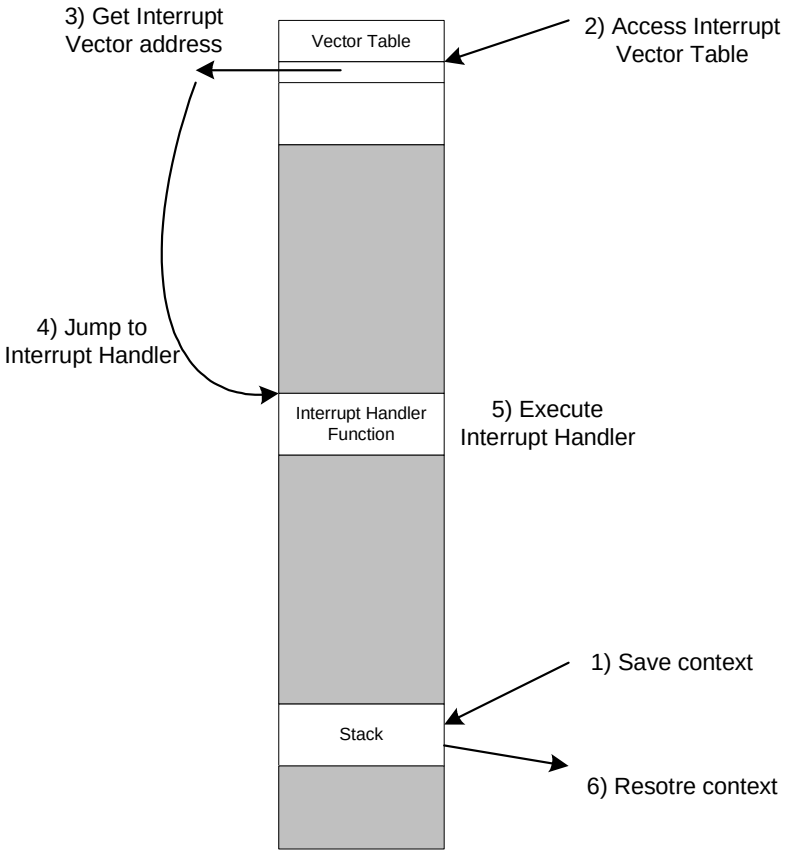


Figure 2.6: Exception 처리 과정

1. **Save context**

   Exception 발생과 관련이 있는 명령이 있는지 확인하여, exception 처리 이후 해당 명령부터 다시 수행하기 위해 현재의 프로세서 상태 및 해당 명령의 주소를 스택 영역에 저장한다.

2. **Access Interrupt Vector Table**

발생된 exception의 종류에 해당하는 인터럽트 벡터 테이블에 접근한다[5]. 인터럽트 벡터 테이블에 대한 자세한 내용은 2.5.2절을 참조하기 바란다.

3. **Get Interrupt Vector address**

   인터럽트 벡터 테이블을 접근하여 exception을 처리하기 위한 인터럽트 핸들러의 함수 포인터를 가지고 온다.

4. **Jump to Interrupt Handler**

   Exception을 처리하기 위한 인터럽트 핸들러 함수로 분기한다.

5. **Execute Interrupt Handler**

   Exception을 처리하기 위한 인터럽트 핸들러 함수를 수행한다. 인터럽트 핸들러에서 사용할 범용 레지스터들은 스택을 통해 보관되어야 하며, exception을 처리한 이후 인터럽트 핸들러 함수를 빠져 나올 때는 저장된 범용 레지스터와 함께 (1)단계에서 저장된 프로세서 상태 및 복구 명령의 주소를 복구시켜야 한다[6].

6. **Restore context**

   인터럽트 핸들러 함수를 수행한 이후 exception 발생 이전의 프로세서 상태로 복구시킨다.

2.5.1 Exception의 종류

1) Reset

프로세서의 초기화 동작을 요구하고자 할 때, 외부에서 프로세서로 입력해 주는 신호를 통하여 발생한다. 이 인터럽트를 받는 경우 프로세서는 초기화를 수행한다. 초기화시 모든 범용 레지스터 및 특수 목적 레지스터는 'unknown'값 또는 '0'의 값으로 변경된다[7].

2) External Hardware Interrupt

외부 하드웨어 모듈에서 요청하는 인터럽트를 의미한다. 현재 상태의 PC, SR 값을 관리자 스텍*(SSP)*에 PUSH한 후 인터럽트 핸들러를 수행한다. 다른 PUSH가 필요한 레지스터들은 인터럽트 핸들러에 의하여 관리되어야 한다. 외부 인터럽트는 다음 두 가지 방식으로 인터럽트 벡터를 지정할 수 있다.

- **Auto-vectored Interrupt**:

  인터럽트 벡터가 지정되어 있는 경우로서, 인터럽트가 발생한 경우 내정되어 있는 인터럽트 벡터로 이동한다.

[5] 인터럽트가 발생하면, 그 인터럽트의 종류에 따라 특정 번호가 주어지며 이를 기반으로 인터럽트 벡터 테이블에 접근한다.

[6] 해당 동작은 인터럽트 핸들러 함수 앞부분에 '#pragma interupt_handler' 를 추가하여 동작되는 부분이다. 자세한 내용은 "Software User Guide"를 참조하기 바란다.

[7] 구현 모델에 따라 변경될 수 있다.

- **Vectored Interrupt**:

  외부 인터럽트 컨트롤러에서 해당 인터럽트의 번호를 넘겨줌으로서, 인터럽트 벡터의 위치를 판별하여 선택된 인터럽트 벡터로 이동한다.

3) Software Interrupt

소프트웨어적으로 발생시키는 인터럽트를 의미한다. 관리자의 자원을 사용자 모드에서 접근하고자 하는 경우 호출하며, 일반적으로 system call을 위하여 사용된다. 현재 상태의 PC, SR 값을 관리자 스텍*(SSP)*에 PUSH한 후, 지정된 인터럽트 번호의 인터럽트 핸들러를 수행한다.

4) Non Maskable Interrupt

Non Maskable interrupt(NMI)는 인터럽트 컨트롤러에서 마스킹 동작을 할 수 없는 인터럽트를 의미한다. 프로세서 외부의 신호를 통하여 발생한다.

5) System Coprocessor Interrupt

메모리 접근 과정에서 발생하는 인터럽트들을 의미한다. Memory Management Unit에 의해 발생되며, 다음과 같은 경우에 발생된다.

- **Access violation**: 관리자 영역으로 지정된 부분을 사용자 모드에서 접근하는 경우
- **Address alignment error**:데이터 크기와 맞지 않는 잘못된 주소 접근의 경우
- **TLB miss**: TLB를 사용하여 가상 주소를 접근하고자 할 때 TLB에 entry가 존재하지 않는 경우[8]

6) Coprocessor Interrupt

보조 프로세서의 접근 과정에서 발생하는 인터럽트를 의미한다. 또한, 이 인터럽트는 연산을 수행하는 보조 프로세서의 연산 결과를 polling하여 발생시킬 수 있다[9]. 보조 프로세서 접근 과정에서 발생하는 인터럽트는 관리자 전용으로 지정된 보조 프로세서를 사용자 모드에서 접근할 때 발생한다.

7) Bus Error

명령어 버스나 데이터 버스의 복구 불가능한 버스의 오류를 의미한다. 목적 주소에 메모리가 존재하지 않는 경우 발생할 수 있다.

- 인터럽트 벡터 테이블 주소가 0x00000010으로 고정되어 있다[10].

[8] 자세한 내용은 5.2를 참조하기 바란다.
[9] EXEC명령에 해당하며 자세한 내용은 2.3.6을 참조하기 바란다.
[10] 자세한 내용은 2.5.2절을 참조하기 바란다.

8) Double Fault

인터럽트가 발생하여 인터럽트 핸들러로 진입하는 과정에서 오류가 발생한 경우를 의미한다. 이 경우 관리자 스택(SSP)의 영역에 문제가 있거나, 인터럽트 핸들러의 위치를 변경시키는 Vector base register의 설정이 잘못되어 있는 경우 발생할 수 있다.

- 인터럽트 벡터 테이블 주소가 0x0000000C로 고정되어 있다[11].

9) Undefined Instruction Exception

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

10) Unimplemented Instruction Exception

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

2.5.2 Interrupt Vector Table

Exception의 처리를 위하여 프로세서는 인터럽트 핸들러 함수를 읽어와야 하는데, 이 인터럽트 핸들러 함수의 주소는 인터럽트 벡터 테이블에 존재한다. AE32000C-Lucida 프로세서의 인터럽트 벡터 테이블은 기본적으로 0x0 번지에 위치하여야 한다. 단, 인터럽트 처리 성능의 향상이나 handler hooking등을 위하여 인터럽트 벡터 테이블의 위치를 변경하는 것이 가능한데, 이는 인터럽트 벡터 테이블을 원하는 부분에 복사하고 2.5.3절에서 설명할 벡터 베이스 레지스터의 값을 변경함으로써 이루어 진다. 단, 몇 가지 주요 인터럽트들은 벡터 테이블의 위치가 변경되더라도 0x0 번지 기반의 벡터 테이블을 이용하여야 한다. 그림. 2.7은 벡터 테이블의 구성을 나타내며, 그림에서 shade 처리 되어 있는 인터럽트들은 벡터 위치를 이동시키는 것이 불가능함을 나타내고 있다.

인터럽트 벡터 테이블을 0x0번지에 위치시키는 것은 프로그램의 배치에 관여하는 loader를 조작함으로써 가능한데, 다음과 같이 loader script에서 벡터 테이블 부분을 지정하면 된다.

```
1  SECTIONS
2  {
3    /* Read-only sections, merged into text segment: */
4    . = 0x0;
5    .text      :
6    {
7      *(.vects) // Location of Interrupt Vector Table
8      ....
```

[11] 자세한 내용은 2.5.2절을 참조하기 바란다.

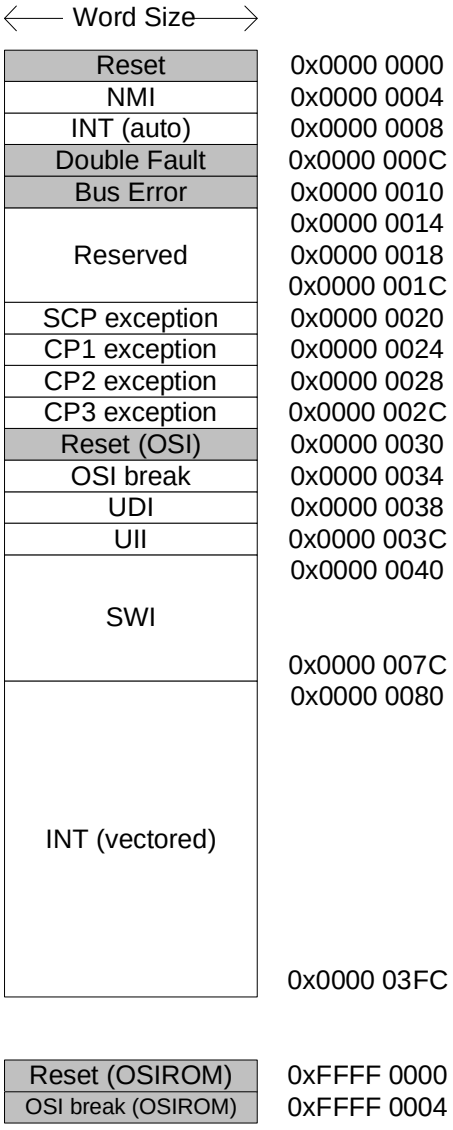


Figure 2.7: 인터럽트 벡터 테이블의 구성

위의 예는 일반적으로 AE32000C-Lucida 프로세서의 firmware 수준의 loader script에서 항상 포함되는 부분으로, 0x0 번지에 .text section을 위치시키되, .text section의 맨 처음에 .vects를 위치 시키라는 의미이다. 이러한 section symbol은 기정의 된 몇 몇 section을 제외하면, 소스 코드 내에 해당 section에 해당하는 부분을 지정해 주어야 한다.

```
1  void (*vector_table[])(void) __attribute__((section (".vects")))= {
2      reset_vector               , /* V00 : Reset Vector         */
3      nmi_autovector             , /* V01 : NMI Vector           */
4      interrupt_autovector       , /* V02 : Interrupt Auto Vector */
5      double_fault_exception     , /* V03 : Double fault Vector   */
6      bus_error_exception        , /* V04 : Bus Error Exception   */
7      NOTUSEDISR                 , /* V05 : Reserved             */
8      NOTUSEDISR                 , /* V06 : Reserved             */
9      NOTUSEDISR                 , /* V07 : Reserved             */
10     coprocessor0_exception     , /* V08 : Coprocessor0 Exception*/
11     coprocessor1_exception     , /* V09 : Coprocessor1 Exception*/
12     coprocessor2_exception     , /* V0A : Coprocessor2 Exception*/
13     coprocessor3_exception     , /* V0B : Coprocessor3 Exception*/
14       ...
15       ...
16       ...
17 };
```

위의 코드에서는 vector_table[]이라는 배열을 .vects section에 포함되도록 지정되었으며, 이 배열은 void (*vector_table[])(void) 형식으로 표현되었으므로, 인자를 지니지 않고, 반환값을 지니지 않는 함수 포인터들의 배열임을 알 수 있다. 즉, 인터럽트 핸들러 함수는 인자나 반환값을 지니지 않는 함수로 작성되어야 한다. 벡터 테이블과 연결되는 인터럽트 함수는 다음과 같이 작성하면 된다.

```
1  #pragma interrupt_handler
2  void nmi_autovector(void) {
3    ...
4    ...
5  }
```

위의 코드에서, #pragma interrupt_handler는 인터럽트 핸들러를 진입시 프로세서의 context를 저장하고, 인터럽트 핸들러 수행 이후 해당 프로세서 context를 복구하기 위한 코드를 자동적으로 추가하는 부분이다. 이 progma 부분은 C 컴파일러에서 다음과 같이 해석된다.

```
1       a60: 3c b4        push  %ml, %mh, %er, %lr
2       a62: ff b0        push  %R0, %R1, %R2, %R3, %R4, %R5, %R6, %R7
3       a64: ff b2        push  %R8, %R9, %R10, %R11, %R12, %R13, %R14, %R15
4
5       ...
6       ...
7
8       a66: ff b3        pop %R8, %R9, %R10, %R11, %R12, %R13, %R14, %R15
9       a68: ff b1        pop %R0, %R1, %R2, %R3, %R4, %R5, %R6, %R7
10      a6a: fc b5        pop %ml, %mh, %er, %lr, %pc, %sr
```

인터럽트 함수의 작성 방법에 대한 자세한 사항은 “Software User Guide”를 참조하기 바란다.

2.5.3 Vector Base

벡터 베이스란 인터럽트 벡터 테이블의 위치를 의미한다. 기본적으로 인터럽트 벡터 테이블은 0x0번지에 위치하며, 벡터 베이스를 변경하여 인터럽트 벡터 테이블의 위치를 이동시킬 수 있다. 단, 앞서 설명했듯이 몇 가지 인터럽트들의 벡터 위치는 이동시킬 수 없으며, 이동될 수 없는 벡터들은 그림. 2.7을 참조하기 바란다. 벡터 베이스는 시스템 보조 프로세서의 12번 레지스터인 벡터 베이스 레지스터에 저장되며, 이 값이 지정된 이후부터 대부분의 인터럽트는 변경된 인터럽트 벡터 테이블로부터 벡터 주소를 읽어온다. 즉, 시스템 보조 프로세서의 12번 레지스터는 프로세서 리셋시 0의 값을 가지며, 레지스터를 설정하게 되면 인터럽트 벡터 테이블의 위치가 변경되는 것이다.

벡터 베이스 레지스터는 벡터 테이블의 접근을 ROM 영역에서 RAM 영역으로 변경하는 경우, 동작의 수행 시간을 단축시키고 인터럽트의 hooking을 가능하게 할 수 있다. 벡터 베이스에서 지정된 벡터 테이블 영역은 가상 메모리를 사용하는 경우 반드시 TLB(Traslate Look-aside Buffer) 내에 존재해야 한다. 이는 AE32000C-Lucida 프로세서의 경우 software managed TLB를 채용하고 있으므로, 벡터 처리 과정 중에 벡터 테이블 접근에서 TLB miss exception이 발생하는 경우 불필요한 double fault가 발생할 수 있기 때문이다.

Vector Base Register의 값을 변경한 것은 단순히 인터럽트 벡터 테이블의 위치가 변경된 것이지 벡터 테이블의 내용인 인터럽트 핸들러 포인터의 배열이 옮겨진 것은 아니다. 인터럽트 벡터 테이블의 내용을 적절한 위치로 복사하거나 생성한 후 Vector Base Register의 값을 변경해야 한다.

2.5.4 Exception Priority

AE32000C-Lucida 프로세서의 인터럽트는 인터럽트들 간의 우선 순위가 존재한다. 프로세서 내부적으로 지니고 있는 우선 순위는 표. 2.13과 같다. 인터럽트 간의 우선 순위는 같은 시점에서 발생한 하나 이상의 인터럽트의 우선 순위를 가리는데 사용되며, 일반적으로는 먼저 요청된 인터럽트가 우선한다.

Table 2.13: 인터럽트 우선 순위

순위	설명	약어
1	Reset	RST
2	Bus Error	BERR
3	Double Fault	DF
4	Coprocessor Exception	CP
5	System-Coprocessor Exception	CP0
6	Non-Maskable Interrup	NMI
7	Software Interrupt	SWI
8	Interrupt	INT
9	Halt	HALT
10	Undefined Instruction Exception	UDI
11	Unimplemented Instruction Exception	UII

Chapter 3

Memory Configuration & Protection

이 장에서는 펌웨어 수준에서 메모리 설정과 보호 기능을 제공하는 Memory Bank Management Unit(MBMU)와 운영 체제(Operating System)수준에서 세밀한 메모리 설정과 보호 기능을 제공하는 UMemory Management Unit(MMU)를 이용하여 각각의 메모리 영역에 대하여 속성을 지정하는 방법과 캐시를 설정하는 방법, Memory Translation을 활성화 시키는 방법에 대하여 설명하도록 한다.

3.1 Memory Management Overview

메모리 관리(Memory management)는 다수의 프로그램이 프로세서 상에서 수행될 때 각각의 프로그램이 사용하는 메모리를 관리/보호하고, 프로그램의 요구에 따라 메모리를 할당/해제하는 일련의 과정을 의미한다. 초기의 내장형 응용(embedded application) 분야에서는 하나의 프로그램 만이 수행되는 형태(firmware level)만이 고려되었으나, 수행하는 작업의 복잡도가 증가함에 따라 RTOS(Real-Time OS)나 일반적인 Desktop에서 사용하던 OS를 사용하게 되는 경우가 많아지게 되었다.

이러한 사용 환경의 변화에 의하여 내장형 마이크로 프로세서도 다수의 프로그램이 동시에 수행되는 경우가 급속히 증가하게 되고 있다. 다수의 프로그램이 수행되는 환경을 지원하기 위해서는 프로그램의 편의성을 고려하여 가상 메모리의 지원 및 프로세스 간의 보호 기능이 필수적인데, 이러한 동작을 수행하는 부분이 메모리 관리 유닛(MMU: Memory Management Unit)이다.

AE32000C-Lucida 프로세서는 사용자의 환경에 따라 가장 적절한 메모리 관리 체계를 선택할 수 있도록 메모리 관리 유닛을 Memory Bank Management Unit과 Translation Lookaside Buffer로 구분하고 있으며, MBMU의 경우 가상 주소 변환이 필요없는 간단한 RTOS에서 사용하기에 적합하며, TLB의 경우 가상 주소 변환 기능을 포함하고 있으며, 메모리 영역에 대한 좀더 세분화된 관리 기능을 지원하여고 강력한 RTOS나 일반 OS에서 사용할 수 있도록 구성되어 있다. 이 장에서는 주로 MBMU에 관하여 설명할 것이며, TLB에 대한 자세한 사항은 5장을 참고하면 된다.

3.2 Memory Bank

AE32000C-Lucida 프로세서는 32비트 프로세서로 최대 4GB의 주소 영역을 가지며, 이 메모리 영역은 뱅크(Bank) 단위로 나뉘어 관리된다. 뱅크란 메모리의 동작 특성을 독립적으로 설정할 수 있는 기본 단위이며, AE32000C-Lucida 프로세서에서는 전체 메모리 영역이 각각 512MB의 크기를 갖는 총 8개의 메인 뱅크(Main-bank)로 나뉘어져 있다. 또한 효율적인 메모리 활용을 위해 추가로 8개의 서브 뱅크(Sub-bank)를 설정할 수 있다.

3.2.1 메인 뱅크 (Main-bank)

그림. 3.1은 프로세서가 접근 가능한 메모리 주소 공간과 나뉘어진 메인 뱅크의 모습을 보여준다.
사용자는 메인 뱅크 레지스터(SCPR9)를 프로그램함으로써 메모리 뱅크의 동작 특성을 설정할 수 있다.

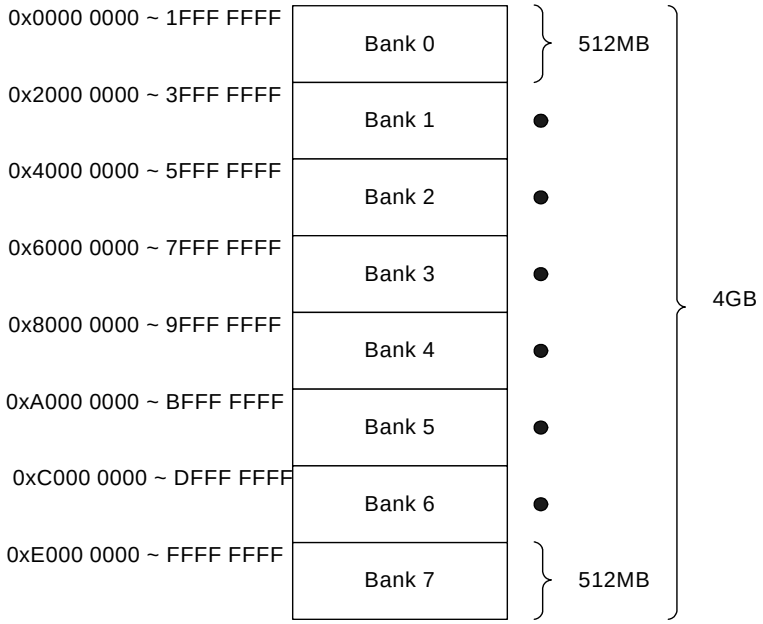


Figure 3.1: AE32000C-Lucida 프로세서의 메모리 관리 구조

3.2.2 서브 뱅크 (Sub-bank)

AE32000C-Lucida 프로세서의 메모리 동작 특성은 기본적으로 512MB 크기의 뱅크 단위로 설정해도 되지만, 내장형 응용 분야에서 이처럼 커다란 메모리를 하나의 동작 특성만을 갖도록 설정하는 것은 메모리 관리의 효율성과 유연성이 저해될 소지가 있다.

가령, 코드가 위치하는 메모리 뱅크 0번을 캐시 동작 가능한 영역으로 설정한 경우, 캐시 메모리에 위치하면 않되는 캐시 초기화 프로그램 또한 뱅크 단위로 설정된 획일적인 동작 특성 때문에 캐시 내에 위치하게 되는 문제가 발생한다.

이러한 치명적인 문제를 해결하기 위해서는 뱅크 단위로 설정된 동작 특성이 뱅크 내의 일부 영역에 대해서는 뱅크 레지스터에 의해 설정된 것과 다르게 설정할 수 있는 매커니즘이 필요하다. 이와 같은 이유로 뱅크 내에서 일부 메모리 영역을 뱅크 속성과 다른 속성을 갖도록 지정할 수 있는 서브 뱅크(Sub-bank)라 불리는 수단이 제공된다. 사용자는 서브 뱅크 인덱스/제어 레지스터(SCPR8)와 서브 뱅크 주소 레지스터(SCPR5)를 프로그램함으로써 서브 뱅크 영역의 범위와 동작 특성을 설정할 수 있다.

3.3 Memory Bank Management Unit의 기능

MBMU는 허가되지 않은 메모리 접근을 차단하는 메모리 보호 기능을 수행한다. 또한 캐시 메모리의 동작 특성에 대한 정보를 생성한다. 이 기능은 주소 변환을 수행하는 TLB 기능이 비활성화(disable) 된 경우에만 유효하며, TLB 기능이 활성화된 경우 TLB 설정에 따라 메모리가 관리된다.

메모리 보호 기능을 통해 허가되지 않은 메모리 접근 혹은 잘못된 주소 접근을 차단할 수 있다. 허가되지 않은 메모리 접근의 경우 access violation exception이 발생하며, 데이터 크기와 맞지 않는 잘못된 주소 접근의 경우 misalign exception이 발생한다.

3.3.1 메모리 접근 권한

U메모리 동작 모드는 사용자 모드와 관리자 모드로 나눌 수 있으며, 메모리 설정 레지스터를 통해 설정할 수 있다. 사용자 모드에서 관리자 모드 전용의 메모리에 접근할 경우, 허가되지 않은 메모리 접근을 차단하기 위하여 access violation exception이 발생한다.

메모리 동작 모드는 메인 뱅크 레지스터(SCPR9), 서브 뱅크 레지스터(SCPR8, SCPR5), TLB 설정 레지스터(SCPR6, SCPR7)를 통해 설정할 수 있다. 설정의 우선순위는 TLB의 설정 > 서브 뱅크의 설정 > 메인 뱅크의 설정 순서이다.

3.3.2 데이터 정렬

AE32000C-Lucida 프로세서는 8비트, 16비트, 32비트 단위로 메모리에 접근할 수 있다. 8비트 메모리 접근에는 아무런 제약이 없으나, 16비트 및 32비트 단위 메모리 접근의 경우에는 메모리 주소가 align에 적합해야 하는 제약이 있다. 16비트 접근의 경우 메모리 주소가 0x0, 0x2, 0x4, 0x6, 0x8, 0xA, 0xC, 0xE로 끝나야 하며, 32비트 접근의 경우 메모리 주소가 0x0, 0x4, 0x8, 0xC로 끝나야 한다. 메모리 주소가 align에 맞지 않는 경우 misalign exception이 발생한다.

3.3.3 캐시 설정

3.3.4 우선 순위

한편, MBMU는 현재 접근중인 메모리 영역의 동작 특성을 결정하기 위하여 뱅크에 설정된 동작 특성과 서브 뱅크에 설정된 동작 설정중에 선택하는 로직을 내포하고 있다.

즉, 메모리를 접근하는 모든 주소는 설정된 서브 뱅크 주소 레지스터들의 값과 비교하여, 주소가 서로 일치하는 경우에는 뱅크에 설정된 동작 특성 대신에 서브 뱅크에 설정된 메모리 동작 설정이 메모리 접근의 동작 특성이 된다.
만약, 해당 뱅크의 설정이 TLB가 enable이면, TLB의 설정이 메모리 접근의 동작 특성이 된다. 즉, 메모리 접근의 우선순위는 TLB의 설정 > 서브 뱅크의 설정 > 뱅크의 설정 순서가 된다.

3.4 Memory Management Register

메모리 보호 유닛은 메모리 접근 권한에 대한 검사 이외에도 캐시 메모리의 동작 특성에 대한 정보를 생성한다. 이 정보는 주소 변환을 수행하는 TLB 유닛이 disable 되어진 경우에 캐시 메모리의 동작 특성을 결정한다. 다음은 메모리 보호 유닛의 구조를 보여준다.

한편, 메모리 동작 모드는 프로세서에서 발생된 주소를 메모리 접근에 직접적으로 사용하는지 여부에 따라 리얼 메모리 모드와 가상 메모리 모드로 나뉠 수 있으며, 이는 메모리 시스템 전체를 구분하는 기준이 된다. 사용자는 가상 주소 - 물리 주소 변환을 수행할지와 메인 뱅크 접근 권한등의 속성을 설정하기 위해서 메인 뱅크 레지스터(SCPR9)를 적절히 프로그램할 수 있다.

3.4.1 메모리 뱅크 레지스터 : SCPR9

메모리 뱅크 레지스터는 니블(Nibble, 4비트) 단위로 하나의 뱅크 설정과 대응하며, 메모리 뱅크별로 설정 가능한 동작 특성은 다음과 같다.

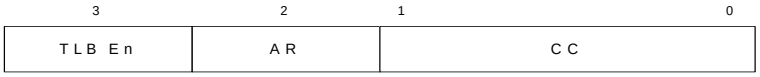


Figure 3.2: SCPR9 Memory Bank Register's Basic Fields

- TLB En (TLB Enable/Disable)
  TLB의 동작 여부를 결정한다.
- AR (Access Right)
  TLB En이 disable인 경우에만 유효하며, 해당 메모리 뱅크의 접근 권한을 설정한다.
- CC (Cache Configuration)
  TLB En이 disable인 경우에만 유효하며, 캐시 메모리와 Write–buffer의 동작 특성을 설정한다.

Table 3.1: SCPR9 register

SCPR9

Bit	R/W	Description	Default Value
31 : 28	RW	Memory Bank 7 Configuration 31 Memory Bank 7 TLB address translation 0 : TLB diable 1 : TLB enable 30 Memory Bank 7 access right (only valid at TLB disable) 0 : Supervisor only 1 : Supervisor / User 29 : 28 Memory Bank 7 Cache Configuration (only valid at TLB disable) 00 : Cache disable 01 : Reserved 10 : Cache enable with write-through 11 : Cache enable with write-back	0000b
27 : 24	RW	Memory Bank 6 Configuration	0000b
23 : 20	RW	Memory Bank 5 Configuration	0000b
19 : 16	RW	Memory Bank 4 Configuration	0000b
15 : 12	RW	Memory Bank 3 Configuration	0000b
11 : 8	RW	Memory Bank 2 Configuration	0000b
7 : 4	RW	Memory Bank 1 Configuration	0000b
3 : 0	RW	Memory Bank 0 Configuration	0000b

3.4.2 서브 뱅크 레지스터 : SCPR5, SCPR8

서브뱅크의 설정시 SCPR8 레지스터를 통해 서브 뱅크의 인덱스를 설정하고, SCPR5 레지스터를 통해 Base address와 offset를 설정하는 순서를 반드시 지켜야 한다. 설정 순서를 지키지 않을시에는 올바르게 서브뱅크가 설정되지 않음에 주의해야 한다. 설정이 올바르지 않은 경우에는 exception이 발생하며, 정상 동작을 보장하지 않는다.
또, 하나의 메모리 뱅크내에 서로 다른 동작 특성을 갖는 여러 개의 서브 뱅크가 존재하는 것을 원칙적으로 허가하지 않으며, 서브 뱅크의 영역이 겹치게 설정하는 것은 허용하지 않는다.

Table 3.2: SCPR5 register

SCPR5

Bit	R/W	Description	Default Value
31 : 12	RW	Sub-bank Base Address Bit 31-12 Base Address = xxxx_x000h	00000h
11 : 0	RW	Sub-bank Size 0000_0000_0000 : 4 KB 0000_0000_0001 : 8 KB 0000_0000_0011 : 16 KB 0000_0000_0111 : 32 KB 0000_0000_1111 : 64 KB 0000_0001_1111 : 128 KB 0000_0011_1111 : 256 KB 0000_0111_1111 : 512 KB 0000_1111_1111 : 1 MB 0001_1111_1111 : 2 MB 0011_1111_1111 : 4 MB 0111_1111_1111 : 8 MB 1111_1111_1111 : 16 MB	000h

Table 3.3: SCPR8 register

SCPR8

Bit	R/W	Description	Default Value
31 : 7	R	Reserved	-
6 : 4	RW	Sub-bank Index 000 : sub-bank 0 001 : sub-bank 1 010 : sub-bank 2 011 : sub-bank 3 100 : sub-bank 4 101 : sub-bank 5 110 : sub-bank 6 111 : sub-bank 7	000b

Table 3.3: SCPR8 register(계속)

Bit	R/W	Description	Default Value
3 : 0	RW	Sub-bank configuration (indexed by bit[6:4] 3 Configration information validity 0 : invalid 1 : valid 2 Access right 0 : supervisor only 1 : supervisor / user 1:0 cache configuration 00 : Cache disable 01 : Reserved 10 : Cache enable with write-through 11 : Cache enable with write-back	0000b

AE32000C-Lucida 프로세서에서는 4KB - 16MB의 크기를 갖는 총 8개의 서브 뱅크 영역을 설정할 수 있다. 메모리 뱅크 관리 유닛은 메모리의 동작 특성을 결정하기 위해서 서브 뱅크의 설정을 메모리 뱅크 설정보다 우선시 한다. 따라서, 뱅크 단위로 설정된 메모리 동작 특성이 서브 뱅크에 설정된 메모리 동작 특성에 의하여 마스킹(masking) 된다. 또한, 서브 뱅크 설정은 메모리가 리얼 메모리 모드로 관리되는 경우에만 즉, 물리 주소로 메모리 접근이 이루어지는 모드에서만 유효한 의미를 갖는다.

3.4.3 뱅크 설정 예제

그림. 3.3은 뱅크와 서브 뱅크의 설정에 대한 예제를 보여준다.

```
1  /* Memory Bank Register */
2    ldi   0x00000070, %r0
3    mvtc  0, 9 // CP0 %r9 setting
4    sync
5
6  /* Sub-Bank 0 Index/Configuration Register */
7    ldi 0x0000000C, %r0
8    mvtc 0, 8 // CP0 %r8 setting
9    sync
10
11 /* Sub-Bank 0 Address Register : Indexed by SCPR8[6:4] */
12   ldi 0x20001000, %r0
13   mvtc 0, 5 // CP0 %r5 setting
```

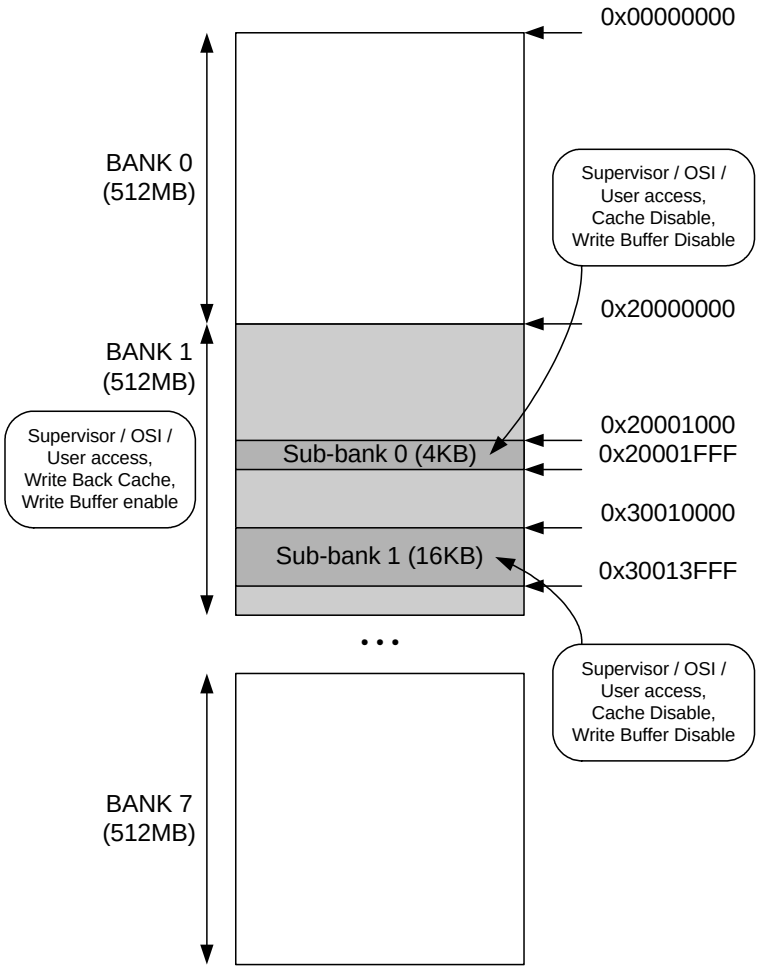


Figure 3.3: 메모리 뱅크와 서브 뱅크의 설정에 관한 예제

```
14    sync
15
16  /* Sub-Bank 1 Index/Configuration Register */
17    ldi 0x0000001C, %r0
18    mvtc 0, 8 // CP0 %r8 setting
19    sync
20
21  /* Sub-Bank 1 Address Register : Indexed by SCPR8[6:4] */
22    ldi 0x30010003, %r0
23    mvtc 0, 5 // CP0 %r5 setting
24    sync
```

예를 들어, 슈퍼 사용자/사용자 모드에서 접근이 허용되고, write back 모드로 동작하는 캐시, write buffer가 동작하는 메모리 영역으로 동작 특성이 설정된 뱅크 내의 일부 영역을 그림. 3.3과 같이 cache disable, write buffer disable 로 설정하기 위해서 사용자는 위 프로그램과 같이 SCPR9, SCPR8, SCPR5 레지스터를 프로그램함으로써 원하는 결과를 얻을 수 있다. 단, 사용자는 뱅크 1 이외의 다른 뱅크의 설정에 관해서는 고려치 않음을 가정한다.

Chapter 4

Cache

본 장에서는 캐시에 대한 전반적인 개요에 대하여 설명하고 AE32000C-Lucida에 사용되는 캐시의 아키텍쳐 및 설정 방법에 대하여 설명하도록 한다. 그리고, 몇가지 예제를 통하여 캐시를 운영할 때 필요한 캐시 초기화(invalidation)와 캐시에 락(lock)을 거는 방법에 대하여 설명하도록 한다.

4.1 Cache Overview

프로세서의 동작 속도에 비하여 일반적으로 사용되는 메모리들의 동작 성능(latency)과 대역폭(bandwidth)은 상대적으로 매우 떨어지므로 시스템의 성능을 저하시키는 주된 원인이 된다. 이는 프로세서의 성능이 높아지면서 더욱 더 부각되고 있는 현상이다. 이러한 메모리 접근에 따른 성능 저하를 해결하기 위하여 AE32000C-Lucida 프로세서에서는 캐시와 버퍼(Buffer) 메모리를 프로세서에 내장하고 있다. 캐시는 프로세서와 메인 메모리 사이에 놓여 있는 전용의 소형 고속 기억 장치이고, 버퍼는 매우 작은 FIFO(First-In-First-Out) 메모리를 말한다[1].

4.1.1 메모리 계층 구조와 캐시 메모리

메모리 시스템은 서로 다른 속도와 크기의 메모리들로 계층 구조로 구성되어 있다. 메모리 계층 구조는 그림. 4.1에 보이는 바와 같이 구성된다. 프로세서와 가까운 부분에 위치한 상위 계층의 메모리는 속도가 빠르지만 용량이 작으며, 하위 계층의 메모리는 속도가 느리지만 용량이 큰 특징을 지니고 있다.

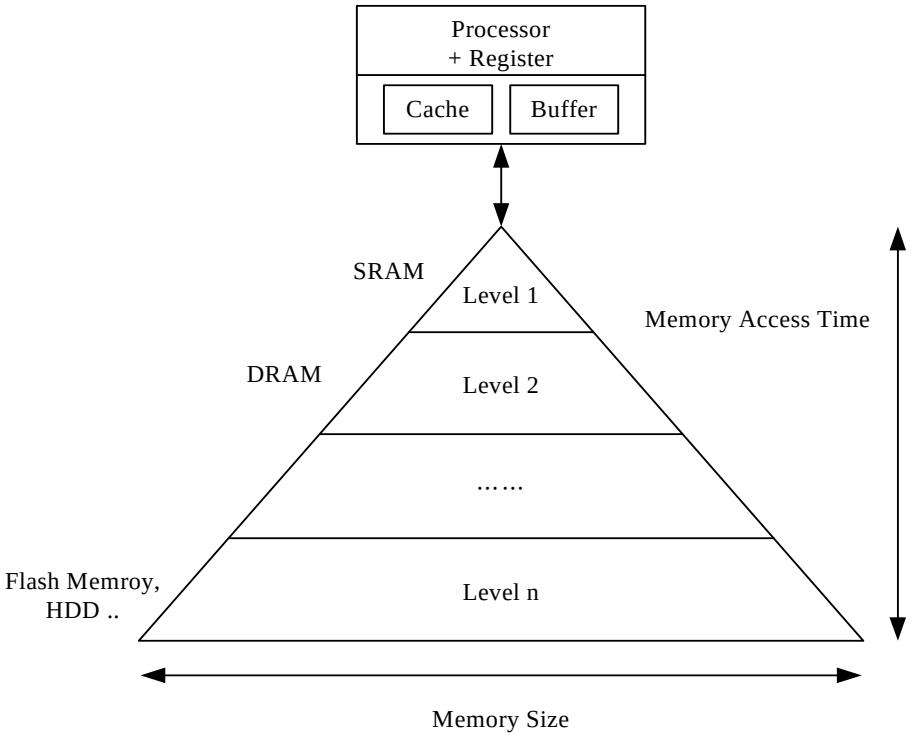


Figure 4.1: 메모리 계층 구조

캐시에 의한 시스템의 성능 향상이 가능한 것은 컴퓨터 프로그램이 지역성의 원칙(principle of locality)을 가진다는 점에 기초하고 있다. 지역성에는 두 가지의 형태가 있는데 어떤 항목이 참조되면 곧바로 다시 참조되기 쉽다는 시간적 지역성(Temporal locality)

[1] 버퍼의 설명은 6장에서 다루도록 한다.

과 어떤 항목이 참조되면 그 근처에 있는 다른 항목들이 곧바로 참조될 가능성이 높다는 공간적 지역성(Spatial locality) 이다.

프로그램에 따라 시간적, 공간적 지역성의 형태는 다르게 나타나지만 대부분의 프로그램/데이터에서 공간/시간적 지역성이 폭넓게 존재하며, 내장형 응용의 경우 프로그램/데이터 모두에 있어서 공간/시간적 지역성이 높게 나타나는 특징을 지니고 있으므로, 캐시의 효율이 비교적 높다[2].

캐시는 그림. 4.2에서 보이는 바와 같이 프로세서와 메인 메모리 사이에 위치하는 메모리로, 프로세서와 가장 가까운 Level 1 Cache(L1 Cache)의 경우 프로세서에서 요구에 적합한 대역폭과 매우 짧은 지연 시간을 지니고 있다. 버퍼는 캐시와 메인 메모리에 존재하는 작은 저장 공간으로서, Write-through 캐시를 운용할 때 메인 메모리 접근에 따른 지연으로 인하여 프로세서 혹은 캐시가 정지하는 것을 방지하는 역할을 한다.

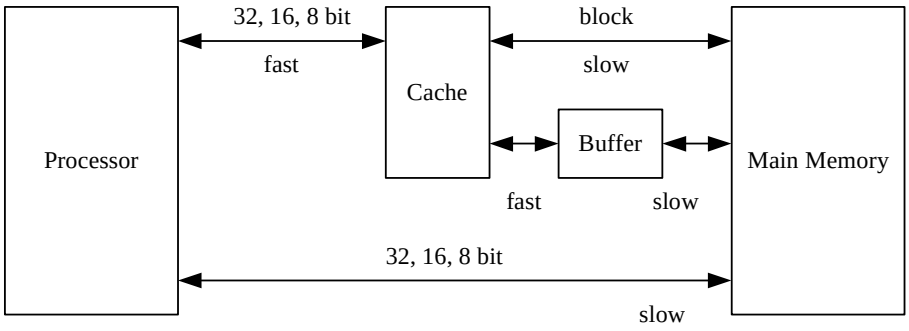


Figure 4.2: 캐시가 갖는 프로세서와 메인 메모리 사이의 관계

4.1.2 Feature

AE32000C-Lucida에 내장된 캐시는 다음과 같은 특징을 지니고 있다.

- 명령어와 데이터가 구분된 하바드 아키텍쳐(Harvard Architecture)
- Virtual index physical taged Address 구조
- Non-Blocking 캐시 접근 지원
- Pseudo-LRU 캐시 라인 교체 알고리즘
- Write-Back 모드와 Write-Through 모드의 쓰기 정책 지원
- 캐시 Locking 기능 지원
- 공정에 독립적으로 합성 가능한 캐시 지원 및 구성
  - `1 ~ 16KB` 캐시 사이즈 지원

[2] 프로그래머는 캐시가 있는 시스템을 운용할 때 반드시 지역성의 특성을 고려하여 프로그램의 제어구조나, 데이터의 형태를 결정하여야 한다.

- Direct-Mapped 또는 2 or 4-way Set-Associate 캐시 지원
- Synchronous SRAM을 사용한 구현

4.2 캐시 아키텍처

일반적으로 프로세서는 폰노이만(Von neumann) 아키텍처와 하버드(Harvard) 아키텍처의 2가지 버스 구조를 가지고 있다. 폰노이만 아키텍쳐는 코어와 메모리 사이에 명령어와 데이터 패스가 하나이고, 하버드 아키텍처는 명령어와 데이터 패스가 분리되어 있다.

폰노이만 아키텍처를 사용하는 프로세서는 명령어와 데이터를 위해 사용하는 하나의 캐시가 있다. 이러한 유형을 통합 캐시(Unified cache)라 한다. 통합 캐시의 메모리는 명령어와 데이터값을 둘 다 포함하게 된다. 하버드 아키텍처는 시템의 성능을 향상시키기 위하여 명령어와 데이터 버스를 분리하여 가지고 있다. 따라서 캐시 또한 명령어 캐시(Instruction cache)와 데이터 캐시(Data cache)가 분할어 있다. 이러한 유형의 캐시를 분할 캐시(Split cache)라 한다. 분할 캐시에서 명령어는 명령어 캐시에, 데이터는 데이터 캐시 안에 저장되어 진다.

AE32000C-Lucida 프로세서는 하버드 아키텍처를 사용하였기 때문에 분할 캐시로 구현되어 있으며, 이는 명령어, 데이터에 대하여 독립적인 제어가 이루짐을 의미한다. 따라서, 캐시를 초기화 시키거나 캐시 락 기능을 사용할 때 사용자는 명령어 캐시와 데이타 캐시에 대하여 각각 제어를 수행해야 한다.

그림 4.4는 캐시 컨트롤러와 캐시 메모리를 포함하는 캐시의 기본적인 아키텍처를 나타내고 있다. 캐시 메모리는 캐시 시스템에서 사용되는 정보를 저장하고 있는 전용 메모리이며, 캐시 컨트롤러는 프로세서에서 발생한 메모리를 이용하여 캐시의 적절한 부분에 대한 접근과 캐시에 대한 관리를 수행하는 부분이다. 본 절에서는 캐시 메모리의 아키텍처에 대해 다룬 다음, 캐시 컨트롤러에 대해 자세히 살펴보도록 한다.

4.2.1 캐시와 메모리 관리 장치

AE32000C-Lucida 프로세서는 가상 메모리 접근 모드(virtual memroy access)를 지원한다. 앞의 3장에서 간략히 설명 되었으나, 가상 메모리 접근 모드에서 프로세서가 운용되는 경우 프로세서에서 발생하는 가상 주소는 실제 물리 메모리에 접근하기 위하여 메모리 변환을 과정을 거쳐야 한다.

캐시의 경우도 가상 메모리 접근에 대한 처리에 따라 변환된 물리 주소를 이용하는 물리 캐시(Physical Cache)와 가상 주소를 이용하는 가상 캐시(virtual cache)로 나뉜다. 논리 캐시는 메모리 변환 이전에 캐시에 접근 할 수 있으므로, 물리 캐시에 비하여 접근 속도가 빠르지만 가상 주소를 사용하기 때문에 프로세스(process)에 따라 중복되는 주소가 존재할 수 있으므로, 프로세스가 변경되어 문맥 변환(context switch)이 일어날때 마다 전체 캐시 영역을 무효화 시켜야 하는 문제와, 동일한 물리 주소주소에 대하여 여러 개의 서로 다른 가상 주소가 사상(mapping)되는 것이 가능하므로, 동일한 물리 주소의 내용이 캐시내의 서로 다른 영역에 존재하고 값이 서로 다르게 운용되는 aliasing 문제가 존재한다.

물리 캐시는 가상 주소를 물리 주소로 변화 해야 하기 때문에 논리 캐시에 비하여 접근 속도가 다소 느리지만 메인 메모리와 같은 주소 사용하여 메인 메모리와 항상 같은 위치에 데이타를 유지하는 장점이 있다.

AE32000C-Lucida 프로세서는 위의 논리 캐시의 빠른 접근이라는 장점과 물리 캐시의 동일 메모리 영역 유지라는 두 가지 장점을 모두 가지는 가상 인덱스와 물리 태그 주

소(Virtual index physical taged address) 방식을 사용함으로써 논리 캐시와 물리 캐시의 장점을 동시에 가지고 있는 특징이 있다. 채용하고 있다. 그림. 4.3은 AE32000C-Lucida 프로세서의 캐시 접근 방식을 나타낸 것으로서, 가상 주소로 캐시에 대한 인덱싱을 수행하고 캐시의 접근이 이루어지는 동안 MMU를 거쳐 나온 물리 주소가 캐시의 태그 비교에 이용되는 그림을 보여주고 있다.

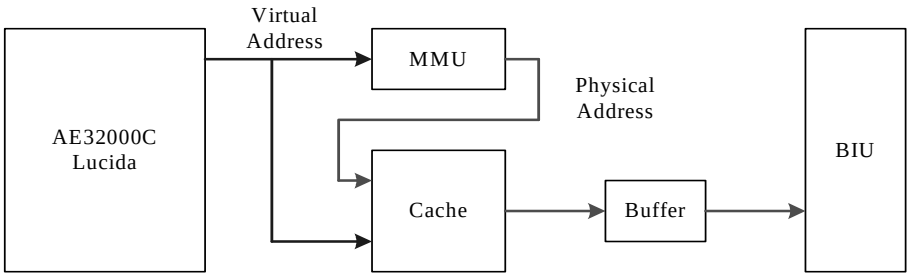


Figure 4.3: 가상 인덱스와 물리 태그 address

가상 인덱스 물리 태그 주소 방식은 구성 가능한 캐시의 way당 크기가 메모리 관리 유닛의 page의 크기로 제한된다는 단점을 지니지만, Desktop이나 Server에서 사용되는 프로세서의 경우에도 프로세스 간의 문맥 교환에 따른 캐시의 미스 처리를 고려하여 일반적으로 L1 캐시의 크기가 16KB를 넘지 않는 경우가 많은 것을 보았을때 큰 제약은 아니다.

4.2.2 캐시 메모리의 기본 아키텍처

간단한 캐시 메모리는 그림 4.4의 오른쪽 부분에 나타나 있다. 캐시 메모리는 캐시 태그, 캐시 데이터, 캐시 상태 영역의 세 부분으로 구성이 되고 이 세부분이 각각의 캐시 라인을 형성하게 된다.

캐시를 운용하기 위해서는 해당 캐시 라인이 어느 메모리 영역의 copy인지 알고 있어야 하는데, 이 부분을 알려주는 정보가 캐시 태그이다. 캐시 데이터는 메인 메모리에서 읽어온 데이터를 저장하는 부분인데, 일반적으로 하나의 캐시 라인은 공간적 지역성을 이용하기 위하여 4-word(16 byte) 이상의 크기로 구성되어 있다. 또한, 캐시의 크기는 캐시가 저장할 수 있는 데이터의 크기로 정의되며, 캐시 태그 및 캐시 상태 영역은 캐시 크기에 포함되어 있지 않다.

캐시 메모리 안에는 캐시의 상태 정보를 나타내는 상태 비트가 포함이 되어 있다. 상태의 비트에는 유효 비트(valid bit), 더티 비티(dirty bit), 락 비트(lock bit)가 있다. 유효 비트는 캐시 라인이 활성화 되어 있음을 표시하는 것으로, 메인 메모리에서 처음에 읽어온 데이터를 포함하고 있고 프로세서에서 사용가능하다는 것을 의미한다. 더티 비트는 캐시 라인이 메인 메모리 안에 저장되어 있는 값과 다른 값을 포함하고 있는지 아닌지를 결정한다. 락 비트는 캐시 라인이 사용자에 의하여 락이 되어 있는지 아닌지를 결정하는

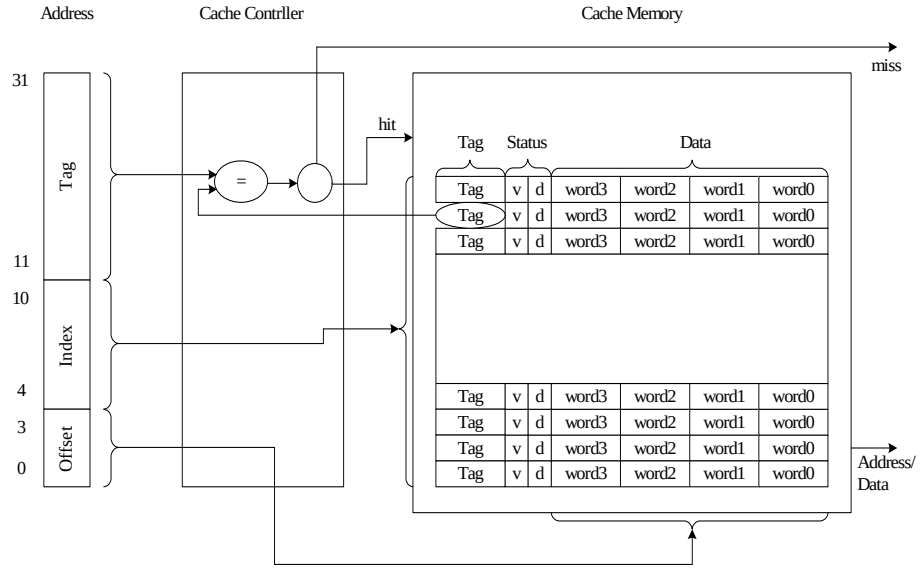


Figure 4.4: 4개의 32비트 워드로 구성된 128개의 라인을 가진 2KB 캐시

것으로 캐시 락 기능이 지원되는 캐시 시스템에서만 사용되어진다.

일반적으로 캐시의 한 라인은 4 word로 구성하지만, 응용 프로그램 특성이나 버스의 형태에 따라 성능 향상을 위하여 8 word(32 byte) 혹은 16 word (64 byte)를 하나의 라인으로 사용하는 경우도 있다. 해당 정보는 일반적으로 프로세서가 제공될 때 같이 제공되나, 이에 대한 추가적인 정보가 필요한 경우에는 담당 엔지니어에게 문의하도록 하라.

4.2.3 캐시 컨트롤러의 기본 동작

캐시 컨트롤러는 캐시에서 미스가 발생한 경우 메인 메모리의 데이터를 적절한 위치로 요구되는 복사하고, 필요한 경우 캐시의 데이터를 메인 메모리로 되쓰기(write-back)하는 하드웨어이다. 캐시의 동작은 소프트웨어의 동작 측면(functionality)에 영향을 주지 않기 때문에(일반적으로 transparent하다고 이야기한다) 소프트웨어 작성시 동작 측면에서 캐시의 존재 유무를 고려할 필요하는 없다. 단지, 캐시의 형태나 동작에 따라 프로그램의 형태에 따라 성능에 차이가 있을 수 있으므로 성능의 최적화를 위해서는 캐시에 대하여 어느 정도 배경 지식을 가지고 있는 것이 필요하다.

캐시 컨트롤러는 프로세서의 읽기 및 쓰기를 위한 메모리 요청을 메모리 컨트롤러에게 전달 되기전에 가로챈다. 이때 요청한 주소를 태그(Tag), 인덱스(Index), 오프셋(Offset)의 세 부분으로 나누어 캐시 영역에 대한 접근과 태그 비교, 적절한 데이터의 위치를 알아내는데 사용한다. 이들 세 부분의 크기는 캐시의 구성이나 크기에 따라 다르며, 그림. 4.4는 2K byte, 1-way(direct mapped) 캐시에서 주소를 태그, 인덱스, 오프셋으로 나눈것을

나타낸다.

컨트롤러는 요청된 코드나 데이터를 포함하고 있는 캐시 메모리의 캐시 라인을 찾기 위해 주소의 인덱스를 사용한다. 인덱스에 의해 선택된 캐시라인은 캐시 태그와 캐시 상태의 정보를 가지고 있는데 캐시 컨트롤로는 이러한 정보를 가지고 캐시 데이터의 사용 가능여부를 결정하게 된다. 만약 캐시 데이터가 사용 가능할 경우 이것을 캐시 적중(hit)이라고 하며, 반대의 경우는 캐시 실패(miss)라고 한다.

캐시 실패시에 캐시 컨트롤러는 메인 메모리에서 실패된 캐시 라인의 모든 정보를 캐시 메모리로 복사하고 요청된 코드나 데이터를 프로세서에게 전달하게 된다. 이러한 캐시 라인을 메인 메모리에서 캐시 메모리로 복사하는 것을 캐시 라인 채우기(cache line fill)라고 한다.

캐시 적중시에 캐시 컨트롤러는 캐시 메모리에 있는 캐시 데이타를 프로세서에 직접 전달한다. 여기서 캐시 데이타에서 정보를 선택하기 위하여 오프셋 값을 사용한다.

4.2.4 캐시와 메인 메모리 사이의 관계

그림. 4.5는 메인 메모리의 일부분이 캐시 메모리 안에 저장되는 위치를 보여주고 있다. 이 그림은 직접 매핑 캐시(Direct mapped cache)로 알려진 캐시 형태를 나타낸 것이다. 직접 매핑 캐시에서는 메인 메모리의 데이터가 캐시 안에 존재할 수 있는 위치가 단 하나만 존재하는 경우를 의미한다.

캐시 메모리는 메인 메모리에 비하여 매우 작은 크기이므로, 메인 메모리상에는 캐시 메모리의 동일한 위치에 매핑될 수 있는 많은 주소가 있다. 그림. 4.5는 하위 주소가 0x124인 많은 데이터들이 같은 캐시의 위치로 매핑될 수 있는 관계를 보여준다.

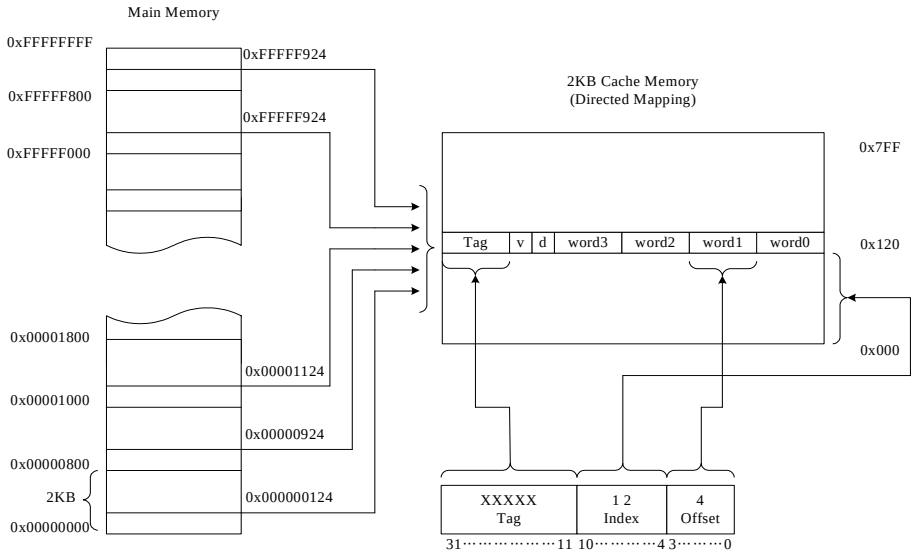


Figure 4.5: 메인 메모리가 직접 매핑 캐시로 매핑되는 방법

1) 직접 사상 캐시(direct mapped cache)

그림 4.4에 소개되어 있는 주소의 3개 필드가 이 그림에도 나와 있다. 인덱스는 0x120 ~ 0x12F로 끝나는 주소를 가진 메모리 안의 모든 값들이 저장되어 있는 캐시의 한 라인을 선택한다. 오프셋은 캐시 라인 안에서 워드/하프워드/바이트를 선택하게 되는데 위 그림의 경우 선택된 라인의 두번째 워드를 선택하고 있다. 태그는 주소의 태그 필드와 비교한 값을 저장하고 있게 된다. 이 예제에서는 캐시 메모리의 한 위치에 대해 메인 메모리 안에 가능한 위치가 이백만 개 있을 수 있다. 주어진 한 순간에는 메인 메모리 안에 가능한 이백만 개의 값 중에서 오직 하나만이 캐시 메모리 안에 존재 할 수 있다.

직접 매핑 캐시(직접 사상 방식 캐시라고도 부른다)의 경우 구조가 비교적 간단하지만, 데이터의 위치가 한 곳으로 고정되어 있으므로 같은 위치로 들어가야 하는 데이터들이 이전에 해당 라인에 존재하는 데이터를 상당히 빠르게 캐시에서 없애는 축출(eviction)이 발생하게 되며, 이를 데이터 간의 충돌(conflict)로 인한 캐시 미스라 부른다. 이런한 현상은 전반적인 캐시의 성능이 떨어지는 현상이 발생할 수 있다. 심지어 그림. 4.6과 같이 프로그램의 동일 루프상에서 서로 다른 주소의 명령어(혹은 데이터)가 같은 캐시 라인에 위치하여 서로를 연속적으로 replace시키는 스래싱(Thrashing) 현상으로 인하여 심각한 성능 저하가 유발되는 경우도 존재한다.

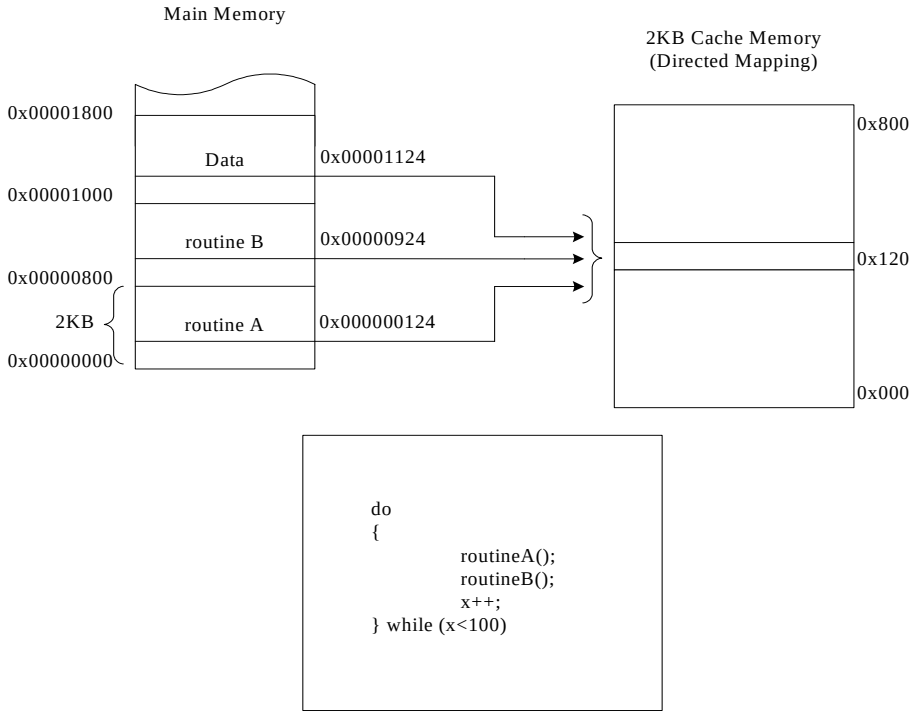


Figure 4.6: 스래싱 : 직접 매핑 캐시에서 두 함수가 교체되는 과정

2) 세트 연상

직접 사상 캐시의 경우 앞에서 이야기 한 것과 같은 데이터간의 충돌로 인한 미스가 많

이 발생하므로, 캐시내에 특정 주소의 데이터가 존재할 수 있는 위치를 여러 곳으로 하는 기법이 사용된다. 이런 위치들을 way라 부르고, 캐시내에 몇개의 위치가 있는지에 따라 n-way 세트 연상 캐시라 부른다. 그림. 4.7은 2KB 크기의 4-way 세트 사상(set associative) 캐시의 예를 보여준다. 그림에서 전체 캐시는 4개의 영역(way)으로 나누어져 있으며, 각각의 부분에 적절히 데이터가 존재할 수 있으므로, 앞에서 설명된 바 데이터 충돌에 의한 미스가 현저히 감소하는 결과를 나타낸다.

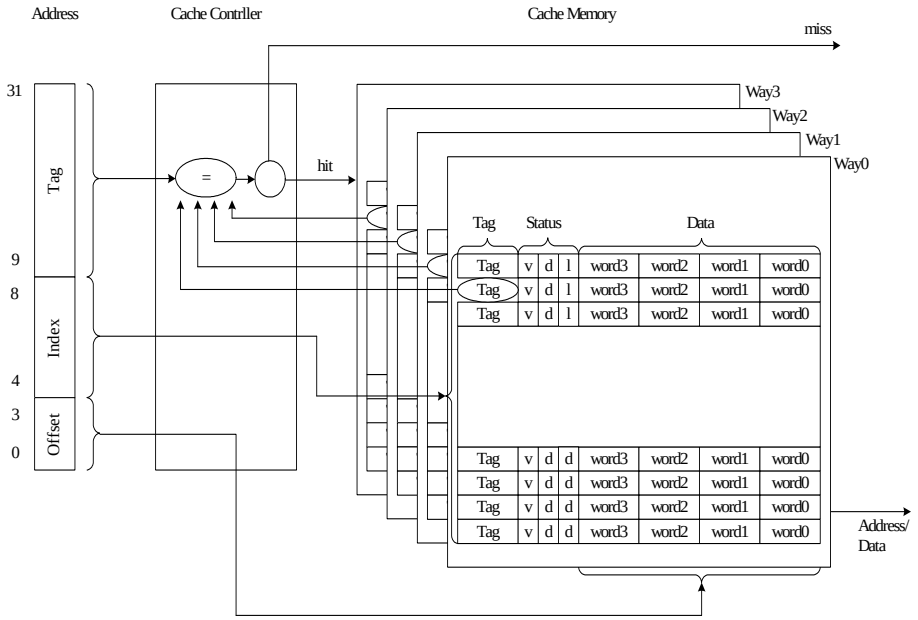


Figure 4.7: 2KB, 4-way 세트 연상 캐시

위의 그림. 4.7에서 인덱스가 가리키는 캐시의 세트를 세트 연상(Set-associative)이라고 한다. 메인 메모리부터의 데이터나 코드는 캐시의 세트에서 4개 way의 어느 캐시 라인에도 할당될 수 있다. 하지만 하나의 세트안에 동일한 코드나 데이타 블록은 하나의 way에만 위치하게 된다.

그림. 4.8은 메인 메모리와 2KB 4-way 세트 연상 캐시의 매핑 관계를 나타낸 것으로, 메인 메모리 안에 있는 어떤 하나의 위치는 캐시 안에 4개의 way중 어떤 곳에나 매핑되어질 수 있다.

일반적으로 캐시에서는 연상도가 증가함으로써 데이터 충돌로 인한 캐시 미스(conflict miss))가 감소하므로 전반적인 성능이 좋아지는 경향이 있다. 극단적으로 한 데이터가 캐시의 어떤 곳으로도 이동할 수 있는 경우를 완전 연상 캐시(Fully associative cache)라 한다. 그러나, 같은 크기의 캐시라 하더라도 캐시의 연상도(associativity))가 증가할 수록 하드웨어의 복잡도와 전력 소모가 증가하므로, 캐시의 사상도는 일반적으로 데스크 탑, 서버 응용이 아닌 경우 4-way 이하로 두는 경우가 많다. AE32000C-Lucida 프로세서는 직접 매핑 캐시, 2-way 세트 연상 캐시, 4-way 세트 연상 캐시의 세가지 형태의 캐시를 지원한다.

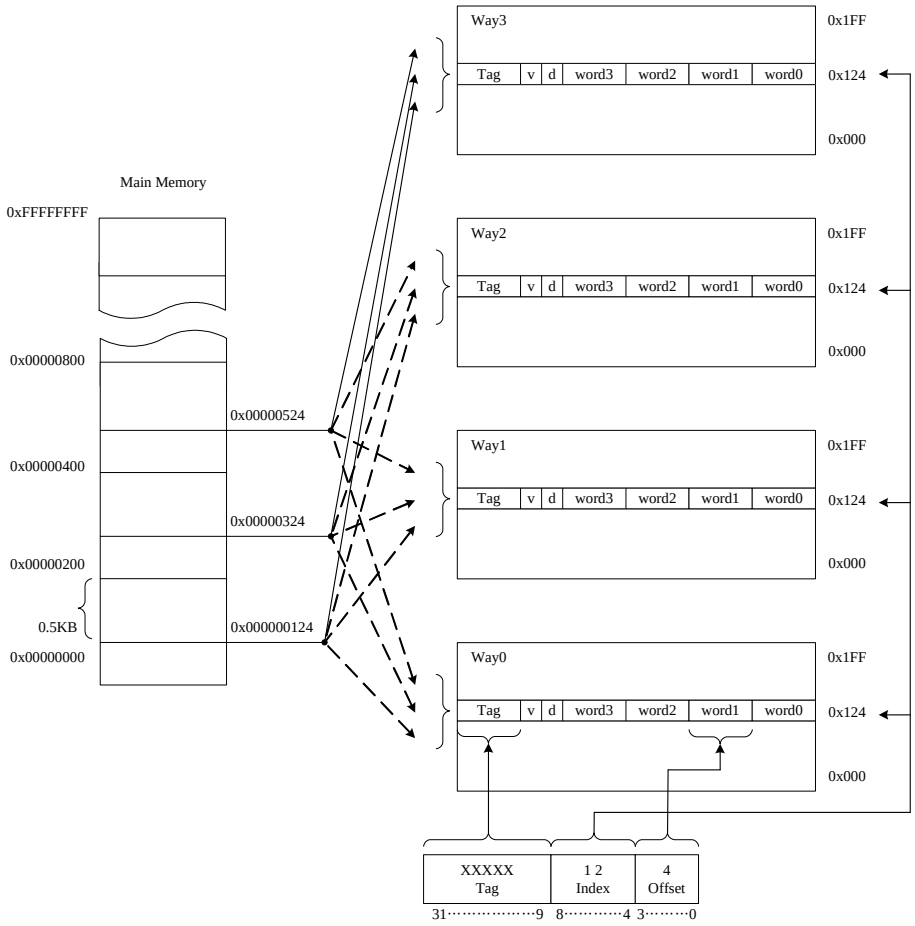


Figure 4.8: 4-way 세트 연상 캐시의 메인 메모리 매핑


캐시의 사상도가 증가하는 경우 캐시 컨트롤러 뿐 아니라 SRAM이 차지하는 면적도 증가한다. 이는 SRAM의 면적이 출력되는 데이터의 크기(data width)에 많은 영향을 받는데, 캐시의 사상도가 증가하는 경우에 동시에 다수의 데이터가 접근되어야 하므로 출력되는 데이터의 크기가 커지는 효과를 가져오기 때문이다. 캐시의 형태에 따른 면적 변화는 별도의 Selection Guide 혹은 담당 엔지니어에게 문의하라

4.2.5 캐시의 효율성

캐시의 효율성을 이야기 할때 사용되는 용어로 적중률(hit rate)과 미스율(miss rate)이 있다. 적중률이란 백불율로 표현을 하며 다음과 같이 나타낸다.

적중률 = (캐시 hit 수 / 메모리 요청 수) x 100

캐시 미스율은 100에서 적중률을 뺀것으로 캐시의 성능을 이야기 할때 적중률 혹은 미스율을 사용한다. 또한 캐시 성능을 이야기 할때 다른 용어로 미스 패널티(miss penalty)가 있는데 미스 패널티란 캐시 미스가 났을 때 메인 메모리에서 캐시로 데이터나 코드를 로드 하는데 걸리는 시간을 이야기 한다.

4.3 Cache Policy

AE32000C-Lucida 프로세서에서 사용되는 캐시의 정책으로는 쓰기 정책(write policy)과 교체 정책(replacement policy)이 있다.

캐시 쓰기 정책은 프로세서가 쓰기 동작을 할 때 데이터를 어디에 저장할지를 결정한다. 교체 정책은 세트 연상 캐시에 적용되는 정책으로 캐시 미스가 발생하는 경우 라인을 채우는데 사용될 세트의 캐시 way를 선택한다.

4.3.1 Write Policy

AE32000C-Lucida 프로세서는 캐시 쓰기 정책으로 연속 기입 방식(writethrough)과 후기입 방식(writeback) 모두 지원한다.

연속 기입 정책은 프로세서가 쓰기 동작을 할때 캐시 적중이면 캐시와 메인 메모리에 모두 데이타를 저장한다. 이 경우에는 캐시와 메인 메모리가 항상 일관성을 유지하게 된다. 하지만 항상 메인 메모리에 쓰기동작을 수행해야 하므로 연속기입 정책은 후기입 정책보다 느리다.

후기입 정책은 유효한 캐시 데이타 메모리에만 저장하고 메인 메모리에는 저장하지 않는다. 따라서 유효 캐시 라인과 메인 메모리는 다른 데이타를 포함 할 수도 있다. 즉, 캐시 라인은 가장 최근의 데이터를 정하고 있는데, 메인 메모리는 아직 업데이트가 되지 않은 오래된 데이터를 가지고 있을 수 있다.

후기입 방식에서 캐시 컨트롤로는 캐시에 데이타를 저장하면 더티 비트를 설정한다. 따라서 케시 컨트롤러는 캐시 메모리와 메인 메모리 사이에 일관성이 유지하기 위하여 더티 비트가 설정된 캐시 라인을 없애버리면 자동으로 메인 메모리에 저장된다.

4.3.2 Replacement Policy

캐시 미스시에 캐시 컨트롤러는 메인 메모리로부터 새로운 정보를 저장하기 위해 캐시 메모리 안에서 새로운 정보로 교체할 캐시 라인을 선택해야 한다. AE32000C-Lucida 프로세서는 캐시 라인 교체 정책으로 Pseudo-LRU 캐시라인 교체 알고리즘을 사용한다.

Pseudo-LUR 교체 알고리즘이란 LRU(Least Recently Used) 정보를 이용하여 인덱스에 의하여 선택된 캐시 세트에서 최근에 사용하지 않은 way의 캐시 라인을 선택하고 교체하는 정책이다. Pseudo-LRU 교체 알고리즘은 캐시 실패 시에 하드웨어로 동작하며, 자동적으로 교체될 캐시 라인을 결정하게 된다.

4.4 캐시 제어 레지스터

AE32000C-Lucida 프로세서에서 캐시의 설정 및 제어는 보조프로세서의 레지스터를 통하여 가능하다.

AE32000C-Lucida 프로세서는 다음과 같이 3가지로 캐시를 설정할 수 있다.

- 캐시 Disabled
- 캐시 Enabled(writethrough 정책)
- 캐시 Enabled(writeback 정책)

캐시의 설정에 영향을 미치는 레지스터는 메모리 뱅크 설정 레지스터(SCPR9), 서브 뱅크 설정 레지스터(SCPR8), TLB 설정 레지스터(SCPR6) 3가지 이다. 캐시를 설정할 때 우선순위는 SCPR6이 가장 높고, SCPR9이 가장 낮아서 SCPR9에서 캐시를 후기입 방식 캐시로 설정을 하여도 SCPR8에서 연속기입 방식 캐시로 캐시가 설정 되어 있다면 SCPR8에 해당되는 물리 주소의 영역은 연속기입 방식 캐시로 캐시가 동작하게된다. (각 레지스터의 설정은 보조프로세서 레지스터를 참고하도록 한다.)

캐시에 영향을 주는 또 다른 보조프로세서 레지스터는 캐시 제어 레지스터(SCPR11)이 있다. SCPR11은 캐시 초기화 루틴과 캐시의 코드나 데이터를 락하는 루틴을 제공하고 있다.

표 4.1은 SCPR11 레지스터의 설명이다.

Table 4.1: SCPR11 레지스터 설명

SCPR11

Bit	R/W	Description	Default Value
31 : 7	W	Target Address[31:7]	-
6 : 4	W	Target Address[6:4] / Target Way	-
3	W	Operation 0 : address based invalidation 1 : way based invalidation	-
2	W	when LOCK = 0, write-back control 0 : without write-back 1 : with write-back if needed	-
	W	when LOCK = 1, Locking procedure 0 : Lock bit read 1 : cache locking	-
1	W	(LOCK) Lock Control 0 : Lock disable 1 : Lock enable	-

Table 4.1: SCPR11 레지스터 설명(계속)

Bit	R/W	Description	Default Value
0	W	Cache Type 0 : Instruction Cache 1 : Data Cache	-

4.4.1 캐시 모드 변경 방법

AE32000C-Lucida 프로세서에서 캐시의 모드 변경은 캐시를 설정하는 것과 같이 보조프로세서 레지스터 SCPR9, SCPR8, SCPR6를 통하여 수행된다. 캐시 모드를 변경하는 과정은 일반적으로 다음과 같이 순서에 의하여 진행된다.

1. 캐시 Disable
2. 캐시 초기화 - 사용자 선택
3. 캐시 모드 변경 및 캐시 Enable

캐시 모드를 변경하기 위해서는 캐시와 메인 메모리 사이에 캐시 일관성을 보장하기 위하여 캐시 설정 레지스터에 접근하기 전에 캐시 초기화를 수행해야 한다.(사용자가 결정) 특히, 후기입 방식 캐시에서 다른 모드로 변경 할때는 캐시 일관성을 보장하기 위하여 캐시 초기화 과정에서 copy-back 모드 캐시 초기화를 사용해야 한다. (캐시 초기화에 관한 내용은 4.5절에 있다.)

```
1   /* Cache Disable */
2   ldi 0x04000004,   %r0
3   mvtc        0,   %r9   // cp0 %r9 setting to cache disable
4   sync
5
6   /* Cache Invalidation
7      Go to cache inval routine
8   */
9   jal <_cache_inval>     // jump to cache inval routine
10  sync
11
12  /* Cache mode change to writeback cache & cache enable */
13  ldi 0x07000007,   %r0
14  mvtc        0,   %r9   // cp0 %r9 setting to cache writeback mode
15  syn
16
```

위의 프로그램은 메모리 뱅크 0과 6을 Write-Back 캐시모드로 변경하는 과정을 나타낸 것으로 캐시 disable, 캐시 invalidation, 캐시 모드 변경 및 enable 의 과정을 확인 할 수 있다.

4.5 Cache Invalidation

캐시 초기화(invalidation)란 캐시의 valid 값을 '0'으로 세팅하여 캐시를 초기화 하는 것을 의미한다. 캐시 초기화 루틴은 캐시를 처음 설정하는 경우, 캐시 모드를 변경할 경우, 캐시의 캐시 락을 해제하는 경우 등 캐시의 상태를 초기화 해야 할 때 사용을 하게 된다.

AE32000C-Lucida 프로세서에서 캐시 초기화 방법으로 주소 단위 캐시 초기화(Address based invalidation)와 Way 단위 캐시 초기화(Way based invalidation)을 지원하고 있다. 캐시 제어 레지스터(SPCR11)를 이용하여 캐시 초기화 및 락 기능을 제어할 수 있다. SCPR11 레지스터에서 유의할 점은 LOCK 비트(`SCPR11[1]`)에 의하여 캐시의 동작이 캐시 초기화 혹은 캐시 락을 결정하게 되는 것이다. SCPR11의 LOCK 비트를 '0'으로 설정하였을때 캐시의 캐시 초기화를 수행하는 일반적인 프로그램 과정이다.

1. 캐시 Disable
2. 캐시 초기화
3. 캐시 설정 및 캐시 Enable

사용자는 캐시 초기화를 수행하기 전에 캐시 *Disable*과 *SCPR11*의 *CPBACK* 비트 설정에 주의를 해야 한다. 캐시 disable은 과정은 캐시 초기화 루틴에서 캐시의 안전성을 보장하기 위한 것이고, CPBACK 비트는 캐시 초기화의 copy-back 모드를 설정하는 비트이다.

Copy-back 모드 캐시 초기화는 데이타 캐시에만 해당이 되는 모드이다. 초기화 할 캐시 메모리의 해당 영역을 메인 메모리로 copy한 이후에 캐시를 초기화 하는 것으로 캐시 메모리와 메인 메모리의 일관성을 유지시켜 준다. 따라서 후기입 방식 캐시를 초기화 할 때에는 copy-back 모드를 꼭 설정해야 한다.

4.5.1 Address Based Invalidation

주소 단위 캐시 초기화(Address based invalidation)는 사용자가 SPCR11에 설정한 주소를 캐시 메모리가 가지고 있으면 설정한 주소의 캐시 라인을 초기화 시키는 것이다. 주소 단위 캐시 초기화는 사용자가 설정한 물리 주소 영역만 초기화를 수행하는 것이기 때문에 캐시의 모든 영역에 대한 초기화가 진행되었음을 보장 할 수 없다. 따라서 캐시의 모든 영역에 대한 초기화는 Way 단위 캐시 초기화를 추천한다.

```
1  /* I-Cache address based invalidation
2     This program will invalidate address 0x10000000 line in I-Cache
3  */
4  push   %r0
5  ldi    0x10000000,   %r0
6  mvtc           0,   %r11
7  sync
8  pop    %r0
9
```

```
10    /* D-Cache address based invalidation
11       This program will invalidate address 0x30000000 line in D-Cache
12    */
13    push   %r0
14    ldi    0x30000001,   %r0
15    mvtc           0,   %r11
16    sync
17    pop    %r0
18
19    /* D-Cache address based invalidation
20       This program will invalidate address 0x30000000 line in D-Cache
21       with copy-back mode
22    */
23    push   %r0
24    ldi    0x30000005,   %r0
25    mvtc           0,   %r11
26    sync
27    pop    %r0
28
```

위 프로그램들은 주소 단위 캐시 초기화의 예제이다. 첫번째는 물리 주소 0x1000000의 명령어 캐시를 초기화 하는 루틴이고, 두번째는 물리 주소 0x3000000의 데이터 캐시를 초기화 하는 루틴이다. 마지막 세번째는 물리 주소 0x3000000의 데이터 캐시를 copy-back 모드로 초기화 하는 루틴이다.

4.5.2 Way Based Invalidation

Way 단위 캐시 초기화(Way based invalidation)는 사용자가 SCPR11에서 설정한 way의 해당 캐시 way를 초기화 시키는 것이다. Way 단위 캐시 초기화는 주소 단위 캐시 초기화와 달리 사용자는 캐시 초기화를 할 물리 주소 영역을 고려하지 않는다. 4way 캐시의 경우 4번의 접근만으로 캐시 전체를 초기화 할 수 있어 일반적으로 캐시 전체를 초기화 할 때 많이 사용된다.

```
1     /* I-Cache way based invalidation
2        This program will invalidate way0 in I-Cache
3     */
4     push   %r0
5     ldi    0x00000008,   %r0
6     mvtc           0,   %r11
7     sync
8     pop    %r0
9
```

```
10      /* D-Cache way based invalidation
11         This program will invalidate way0 in D-Cache
12      */
13      push   %r0
14      ldi    0x00000009,   %r0
15      mvtc           0,   %r11
16      sync
17      pop    %r0
18
19      /* D-Cache way based invalidation
20         This program will invalidate way0 in D-Cache
21         with copy-back mode
22      */
23      push   %r0
24      ldi    0x0000000D,   %r0
25      mvtc           0,   %r11
26      sync
27      pop    %r0
28
```

위 프로그램들은 Way 단위 캐시 초기화의 예제이다. 첫번째는 명령어 캐시의 way0를 초기화 하는 루틴이고, 두번째는 데이터 캐시의 way0를 초기화 하는 루틴이다. 마지막 세번째는 데이터 캐시의 way0를 copy-back 모드로 초기화 하는 루틴이다.

4.6 Cache Lock

캐시 락(cache lock)이란 프로그램에서 동작 속도에 민감한 코드와 데이터를 캐시에 로드한 후 캐시의 교체에서 면제되도록 하는 방법(scheme)을 의미한다.

캐시 내에 locking된 코드나 데이터는 캐시의 메모리 안에 항상 저장되어 있기 때문에 빠른 시스템의 성능을 제공하게 되고, 캐시 미스 패널티(miss penalty)에 의한 예기치 못한 실행 시간의 문제점을 피할 수 있다.

하지만 캐시 락을 사용하게 되면 캐시의 영역이 줄어 들기 때문에 사용 가능한 캐시의 크기는 줄어든다. 일반적으로 캐시 락을 사용하는 코드나 데이터는 interrupt vector table, interrupt service routine, critical algorithm of system, global variable 등이 있다.

AE32000C-Lucida 프로세서에서 캐시 락하는 방법으로 주소 단위 캐시 락(Address based lock)과 Way 단위 캐시 락(Way based lock)을 지원하며, 다른 상용 프로세서와는 달리 캐시 락 과정에서 1)캐시 초기화, 2)캐시 락, 3)데이터 로드 과정을 한번에 수행한다. AE32000C-Lucida 프로세서에서 캐시 락 과정은 다음과 같다.

1. 캐시의 락 상태 체크
2. Disable Interrupt
3. 캐시 Disable
4. 캐시 락 (초기화, 락, 데이타 load 포함)
5. 캐시 Enable
6. Enable Interrupt

AE32000C-Lucida 프로세서에서 캐시 락은 세트 연상 캐시에서만 지원이 되며[3] 다음과 같은 제약 사항이 있다. 1)캐시의 모든 way를 캐시 락 할 수 없고, way 단위 캐시 락은 최대 way의 1/2 까지 지원한다. 2)주소 단위 캐시 락은 모든 way를 락 할 수 없기 때문에 락 가능여부를 사용자는 항상 확인 하여야 한다.

4.6.1 Lock Status Check

AE32000C-Lucida 프로세서는 캐시의 락 상태 정보 확인 한 후 캐시 락 가능 여부를 사용자가 판단 하여야 한다. 특히 AE32000C-Lucida 프로세서는 모든 way에 캐시 락을 할 수 없기 때문에 하드웨어 구성에 따라 사용자의 주의가 필요하다.

캐시의 캐시 락 상태를 확인 하기 위해서는 SCPR11, SCPR3, SCPR4 레지스터에 접근해야 한다. SCPR11 레지스터는 캐시의 락 상태 정보에 접근하는 역할을 한다. (LOCK = '1'로 LOCKPRO = '0'으로 설정) SCPR3과 SCPR4 레지스터는 사용자에게 캐시의 락 상태를 알려주는 레지스터로 설정 방법은 보조프로세서 레지스터를 참고하도록 한다.

[3] direct mapped 캐시에서는 way가 하나 뿐이기 때문에 cache lock 기능을 지원하지 않는다.

제시될 캐시 락 체크 프로그램의 예문에서 보면 SCPR11 레지스터로 캐시의 락 상태 정보에 접근한 이후 SCPR3과 SCPR4 레지스터를 통하여 캐시의 락 정보를 읽어 오는 것을 확인 할 수 있다. 사용자는 이렇게 읽어 들인 캐시의 락 정보를 통하여 원하는 영역의 락 여부를 판별하여 캐시 락 루틴을 수행하면 된다.

아래 프로그램의 예시는 물리 주소 0x1000000의 명령어 캐시 락 체크 루틴과 물리 주소 0x3000000의 데이터 캐시 락 체크 루틴이다. 그리고 캐시 락 정보 저장하기 위하여 레지스터 8을 할당 하였다.

```
1   /* I-cache add lock check
2      This program will store address 0x10000000 lock data
3      in I-cache to register 8
4   */
5   sync
6
7   push   %r0
8   ldi    0x10000002,   %r0
9   mvtc           0,   %r11   // cp0 %r11 setting to add lock check
10
11  ldi         0x500,   %r0
12  mvtc           0,   %r3    // cp0 %r3 setting to access I-cache lock data
13
14  mvfc           0,   %r4    // cp0 %r4 setting to read lock data
15  st           %r0,   %r8    // copy lock data to register 8
16
17  sync
18  pop    %r0
19
20
21  /* D-cache add lock check
22     This program will store address 0x30000000 lock data
23     in D-cache to register 8
24  */
25  sync
26
27  push   %r0
28  ldi    0x30000003,   %r0
29  mvtc           0,   %r11   // cp0 %r11 setting to add lock check
30
31  ldi         0x501,   %r0
32  mvtc           0,   %r3    // cp0 %r3 setting to access D-cache lock data
33
34  mvfc           0,   %r4    // cp0 %r4 setting to read lock data
35  st           %r0,   %r8    // copy lock data to register 8
```

```
36
37      sync
38      pop    %r0
39
```

아래 프로그램의 예시는 명령어 캐시와 데이터 캐시의 way에 대한 락 체크 루틴이다. 캐시 락 정보는 레지스터 8을 사용하여 저장하였다.

```
1       /* I-cache way lock check
2          This program will store way lock data in I-cache to register 8
3       */
4       sync
5
6       push   %r0
7       ldi    0x0000000A,   %r0
8       mvtc           0,   %r11   // cp0 %r11 setting to way lock check
9
10      ldi         0x500,   %r0
11      mvtc           0,   %r3    // cp0 %r3 setting to access I-cache lock data
12
13      mvfc           0,   %r4    // cp0 %r4 setting to read lock data
14      st           %r0,   %r8    // copy lock data to register 8
15
16      sync
17      pop    %r0
18
19
20      /* D-cache way lock check
21         This program will store way lock data in D-cache to register 8
22      */
23      sync
24
25      push   %r0
26      ldi    0x0000000A,   %r0
27      mvtc           0,   %r11   // cp0 %r11 setting to way lock check
28
29      ldi         0x501,   %r0
30      mvtc           0,   %r3    // cp0 %r3 setting to access D-cache lock data
31
32      mvfc           0,   %r4    // cp0 %r4 setting to read lock data
33      st           %r0,   %r8    // copy lock data to register 8
34
35      sync
```

```
36      pop     %r0
37
```

위 캐시 락 체크 예제 프로그램에서는 캐시 락 정보 저장하기 위하여 레지스터 8을 할당 하였다. 사용자는 예제 프로그램처럼 캐시 락 정보를 저장한 이후 분석을 하여 캐시의 락 가능 여부를 판별하여야 한다.

캐시 락 정보는 8bit로 나타내어 진다. [4]

아래의 표 4.2는 사용자가 읽어 들인 락 상태 정보에 따른 캐시 락 가능 여부를 간략하게 나타낸 표이다.

Table 4.2: 캐시 락 상태 정보 분석

Lock Mode	Cache Type	Lock check Data	Lock Enable
Address based lock	2way	00000001	Disable
		00000010	Disable
		else	Enable
	4way	00001110	Disable
		00001101	Disable
		00001011	Disable
		00000111	Disable
		else	Enable
Way base lock	2way	00000000	way0 enable
		else	Disable
	4way	0000xx00	way0, 1 enable
		0000xx01	way1 enable
		0000xx10	way0 enable
		00001101	Disable
		00001110	Disable
		else	Disable

4.6.2 Address Based Lock

주소 단위 캐시 락(Address based lock)은 사용자가 지정한 물리 주소 영역을 락하게 된다. 주소 단위 캐시 락은 사용자가 지정한 물리 주소 영역이 캐시 메모리에 있으면 해당 way의 캐시 라인을 캐시 락하게 된다. 만약 사용자가 설정한 물리 주소 영역이 캐시 메모리에 없으면 오림차순 순으로 설정한 물리 주소의 해당 캐시 라인을 캐시 락한다.

[4] AE32000C-Lucida 프로세서는 최대 8way의 캐시로 하드웨어를 구성할 수 있다. 각 way는 1bit의 캐시 lock 정보를 가지므로 사용자는 캐시 lock 정보를 8bit로 획득하게 된다. 0way는 0번째 비트에 7way는 7번째 비트에 정보가 실린다. 현재는 4way 까지 지원

```
/* I-Cache address based invalidation
   This program will invalidate address 0x10000000 line in I-Cache
*/
sync

push   %r0
ldi    0x10000006,   %r0
mvtc           0,   %r11
sync
pop    %r0

/* D-Cache address based invalidation
   This program will invalidate address 0x30000000 line in D-Cache
*/
sync

push   %r0
ldi    0x30000007,   %r0
mvtc           0,   %r11
sync
pop    %r0

```

위 프로그램들은 주소 단위 캐시 락의 예제이다. 첫번째는 물리 주소 0x1000000의 명령어 캐시를 락하는 루틴이고, 두번째는 물리 주소 0x3000000의 데이터 캐시를 락하는 루틴이다.

4.6.3 Way Based Lock

Way 단위 캐시 락(way based lock)은 사용자가 설정한 way에 설정한 물리 주소부터 way 크기 만큼 영역을 락하게 된다. 이때 주의해야 할 점은 way 단위 캐시 락은 락을 할 데이타의 크기가 하드웨어적으로 항상 way 크기 만큼 고정되어 있다는 것이다.

예를 들어 8KB 4way 세트 연상 캐시의 경우 한 way의 크기가 2KB 이기 때문에 way 단위 캐시 락을 하게 되면 한번에 2KB의 물리 주소 영역이 캐시 락이 된다.

```
/* I-Cache address based invalidation
   This program will invalidate address 0x10000000 line in I-Cache
*/
sync

push   %r0
```

```
7     ldi    0x1000000E,   %r0
8     mvtc           0,   %r11
9     sync
10    pop    %r0
11
12    /* D-Cache address based invalidation
13       This program will invalidate address 0x30000000 line in D-Cache
14    */
15    sync
16
17    push   %r0
18    ldi    0x3000000F,   %r0
19    mvtc           0,   %r11
20    sync
21    pop    %r0
22
```

위 프로그램들은 way 단위 캐시 락의 예제이다. 첫번째는 물리 주소 0x1000000부터 2KB 만큼 명령어 캐시 way0를 락하는 루틴이고, 두번째는 물리 주소 0x3000000부터 2KB 만큼 데이터 캐시 way0를 락하는 루틴이다.

Chapter 5

TLB

이 장에서는 사용자가 AE32000C-Lucida 프로세서에서 가상 주소를 이용하고자 할때 알아야 하는 사항들에 대하여 다루도록 한다.

- Memory Management Unit을 통한 가상 주소의 물리 주소로의 주소 변환 과정과 메모리 관리 기능에 대하여 설명한다.
- AE32000C-Lucida 프로세서는 가상 주소의 물리 주소로의 변환을 가속하기 위하여 TLB(Translated Lookaside Buffer)를 지원하며, MMU와 동일하게 TLB의 설정 및 관리는 운영체제(Operating System)에서 담당한다. 이 장에서는 운영체제에서 AE32000C-Lucida 프로세서의 TLB를 운용할 때 고려해야 하는 사항들과 등에 대해서 설명한다.

5.1 TLB Overview

TLB란 Translation Look-aside Buffer의 약자로서, O/S가 만드는 페이지 테이블의 일부를 보관하는 캐시이다. 페이징(Paging)은 개념이 프로그래머에게 가상 메모리를 사용하고, O/S에서 메모리를 관리하는데 상당한 편의성을 제공하지만, 가상 주소로 적절한 페이지를 찾아 물리 주소로 변환을 위하여 페이지 디렉토리와 페이지 테이블과 같은 구조에 대한 접근이 요구된다. 이러한 구조에 대한 접근 자체가 메모리 접근을 유발하므로, 순수하게 소프트웨어적으로 처리되는 경우 상당한 성능 저하가 일어나므로, 사용하고 있는 프로세스와 관련된 페이지 테이블을 미리 별도의 메모리로 캐싱하는 방법을 사용하게 되고, 이때 사용되는 메모리를 TLB라 부른다. TLB를 관리하는 방법은 TLB miss가 발생한 경우에 O/S가 필요한 페이지와 관련된 페이지를 TLB에 채워놓는 경우(software managed TLB)가 있고, TLB miss가 발생한 경우 하드웨어적으로 page table walking을 수행하여 필요한 페이지를 TLB로 가지고 오는 기법이 있다.

1) MMU의 구성

AE32000C-Lucida 프로세서의 MMU는 메모리를 페이지(page) 단위로 관리하는 paging 기법을 통하여 메모리를 관리한다. AE32000C-Lucida는 4KByte 단위의 페이지를 이용하며, 모든 메모리 주소 변환이나 권한 설정, 접근 보호는 페이지 단위로 이루어짐을 의미한다. AE32000C-Lucida 프로세서의 MMU는 TLB(Translation Lookaside Buffer)와 접근 권한 제어 유닛으로 구성되어 있다. TLB는 O/S에서 구성한 페이지 테이블의 일부를 보관하는 일종의 캐시로, AE32000C-Lucida는 TLB의 관리를 소프트웨어로 직접 수행하는 software managed TLB의 형태를 취하고 있다.


AE32000C-Lucida 프로세서의 경우 4KB 단위의 페이지를 사용하고 있으나, AE32000C 프로세서 아키텍쳐 자체는 시스템 보조 프로세서(System Co-processor)의 형태에 따라 4KByte 페이지와 256KByte 페이지를 동시에 지원할 수 있다. 일반적인 O/S 작성자는 포팅을 고려하여 4KB 페이지 단위로 메모리 관리 기능을 작성하여야 한다.

Software Managed TLB 본 프로세서에서 채용한 TLB는 Software Managed TLB의 형태를 취하고 있으므로, TLB miss exception이 발생한 경우 운영 체제는 해당 페이지의 정보를 Page Table에서 TLB로 옮기는 동작이 필요하다.

Processor Operation Mode AE32000C-Lucida는 관리자 모드(Supervisor Mode)와 사용자 모드(User Mode)로 구분하여 페이지의 접근을 제한하는 기능을 제공한다. 또한 Process ID(PID)를 사용하여 프로세스 단위로 접근을 제한하는 기능을 제공한다.

AE32000C-Lucida에 적용된 MMU는 메모리 동작 모드에 따라 사용되는 유닛에 차이가 있다. AE32000C-Lucida에 메모리 동작 모드는 가상 메모리의 사용 여부에 따라 물리 메모리 접근 모드와 가상 메모리 접근 모드로 구분된다. 물리 메모리 접근 모드는 주소

변환 사상이 일어나지 않는 모드이며, 가상 메모리 접근 모드는 프로세서에서 생성된 주소를 물리 메모리로 변환하는 주소 변환이 일어나는 모드이다. 이러한 메모리 동작 모드의 설정은 메모리 뱅크 별로 설정 가능하다. 메모리 뱅크 설정의 자세한 사항은 3.2절을 참고하기 바란다.

물리 메모리 접근 모드 물리 메모리 접근 모드에서 프로그래머가 사용할 수 있는 메모리 영역은 실제 제공되는 물리 메모리의 크기에 따라 달라지나, 일반적으로 각 뱅크에 대한 주소의 구분은 프로그램의 재 사용성을 높이기 위하여 가상 메모리 사용시와 동일한 4GB 영역으로 잡는 것이 일반적이다. 물리 메모리 접근 모드는 메모리 뱅크 및 서브 뱅크의 설정에 따라 메모리 접근 권한이나 캐시 메모리의 동작을 설정할 수 있는 메모리 보호 기능을 제공한다.

가상 메모리 접근 모드 가상 메모리 접근 모드에서 프로그래머는 실제 시스템에서 물리 메모리의 구성에 관계 없이 4GB 주소 영역을 자유롭게 사용할 수 있다[1]. 가상 메모리 접근 모드는 뱅크 단위로 해당 뱅크에 대한 가상 메모리를 사용할 것인지, page의 크기를 4Kbtye 또는 256 Kbyte로 할 것인지에 대하여 설정하여 사용할 수 있다(일반적인 AE32000C-Lucida의 경우 4KByte page만 사용 가능하다).

[1] O/S에 따라 일부 제약을 가지는 경우도 있다.

5.2 TLB의 동작

AE32000C-Lucida에 사용되는 TLB는 소프트웨어의 관리를 받는 형태의 TLB이다. 따라서 운영체제는 프로세스의 변경시에 관련된 페이지를 TLB에 넣어줌으로써 처리를 수행할 수 있으며, TLB miss가 발생한 경우에도 이를 처리할 수 있다.

AE32000C-Lucida의 TLB는 접근 속도와 hit 율을 높이기 위하여 2-level로 구성되어 있고, 1st level TLB의 경우 하드웨어적으로 관리하고 있으므로 운영 체제에서 이 부분을 따로 관리할 필요는 없으며, 이후에 설명되는 것은 모두 소프트웨어가 처리해 주어야 하는 명령어/데이터 버스에 대해 구분되는 separated 형태의 2nd level TLB 부분에 대해서만 설명하도록 한다.

AE32000C-Lucida의 TLB는 128 entry 4 way set associative separated 형태로 구성되어 있다. 운영체제의 메모리 관리 부분은 수행해야 하는 프로세스를 원활히 지원하기 위하여 TLB에 접근해야 한다.

AE32000C-Lucida에서 소프트웨어적으로 TLB를 관리하기 위하여 제공하는 기능과 관련 동작은 다음과 같다.

TLB enable/disable TLB를 사용할 지 여부를 나타낸다. 만일 지정된 영역에 TLB disable되어 있다면, 주소 변환 사상이 수행되지 않으며, 이는 가상 주소와 물리 주소가 동일한 값을 가짐을 의미한다. MBMU를 통하여 지정되어야 한다.

TLB Invalidation TLB의 하나의 특정 entry만을 선택하여 무효화(invalidation)하는 기능을 제공한다.

TLB enable시의 동작 가상 주소를 이용하여 메모리를 접근하는 경우 MMU는 TLB를 접근하여 TLB hit/miss를 판별한다. 만약 TLB hit라면 가상 주소에 대응하는 TLB entry의 물리 주소와 캐시 설정, protection 정보를 캐시 컨트롤러와 메모리 관리 유닛에 각각 전달한다. 만일 TLB miss라면 정수 코어에 TLB miss exception임을 알려서 운영 체제의 커널에서 TLB miss에 대한 처리(적절한 페이지를 가지고 와서 TLB에 넣는 과정과 page swap)를 수행한다.

TLB Miss 다음과 같은 경우는 TLB miss로 처리되어 exception이 전달된다.

- TLB entry miss: 원하는 데이터의 주소 정보가 TLB entry에는 존재하지 않으며, 물리 메모리의 페이지 테이블에는 존재하는 경우이다. TLB entry에 대한 교체가 필요하다.
- Page fault: 원하는 데이터의 주소 정보가 TLB entry에 존재하지 않으며, 물리 메모리의 페이지 테이블에도 존재하지 않는 경우로써, 해당 데이터가 디스크와 같은 보조 기억 장치에 존재하는 경우이다. 보조 기억 장치로부터 데이터를 읽어서 물리 메모리로 옮기고, 페이지 테이블 및 TLB의 entry를 교체하면 된다.

TLB Miss Handling TLB miss가 발생한 경우 프로세서는 이를 시스템 보조 프로세서 예외(system coprocessor exception. 5.4절 참조)로 받아들이게 된다. 시스템 보조 프로세서 예외는 시스템 보조 프로세서의 SCPR15의 상태를 통하여 1) 어떤 종류의 exception이 2) 어디에서 발생했는지 알아낸다. 만일 TLB miss에 관련된 것이라면, TLB miss가 발생한 가상 주소를 확인하여 이를 통하여 page table walking을 수행해야 하는데, TLB miss가 발생한 가상 주소는 데이터 접근에 의한 miss인 경우 GAPgeneral access point register인 SCPR3와 SCPR4를 이용하여 찾을 수 있으며, 명령어 접근에 의한 가상 주소는 `push %pc` 하여 확인 할 수 있다.

Page Table Walk TLB miss를 발생한 경우 해당 가상 주소를 바탕으로 물리 주소의 page table에 접근하여 page fault인지 그렇지 않은지를 확인한다. 만약 page fault가 아닌 경우 miss가 발생한 TLB entry에 page table의 정보를 기록하는 TLB entry 교체 작업을 수행하면 된다. 만약 page fault인 경우 physical disk에 존재하는 데이터를 physical memory와의 swapping(paging) 하는 동작을 수행하고 관련 정보를 page table과 miss가 발생했던 TLB entry에 기입한다.


AE32000C-Lucida는 각 프로세스마다의 가상 주소를 잘 지원하기 위하여 어떤 프로세스의 가상 주소인지 확인 할 수 있도록 TLB상에 process identifier register를 두고 있다. 운영 체제에서는 프로세스 ID를 적절하게 설정하여 프로세스 문맥 교환에 따른 TLB miss 발생을 최소화 할 수 있다.

다음은 TLB 접근 및 교체 과정에 대한 일반적인 pseudo code이다.

```
TLB Lookup :
    TLB enable = Memory Bank Unit (Virtual address)
    if (TLB enable == 1)
      if (TLB address/id match)
        if (write at read only region)
          After TLB miss exception occurs, Kernel TLB miss handling.
        else if ( user program try to access supervisor region)
          After TLB miss exception occurs, Kernel TLB miss handling.
        else if (data write cycle)
          Set TLB.dirty.
        Set TLB.accessed.
        Translate virtual address to physical address in TLB entry.
      else
        After TLB miss exception occurs, Kernel TLB miss handling.
    else
      Physical address is same with virtual address

Kernel TLB miss handling :
```

```
19    Check where TLB miss occurs in instruction fetch or data access?
20    Page Table Walking
21    if (page reside in physical memory)
22      Manages missed TLB entry
23      Return TLB miss program
24    else
25      Swapping(paging) between physical memory and physical disk
26      Manage TLB entry
27      Return TLB miss program
```

5.3 TLB register

TLB register에는 크게 TLB index register와 TLB virtual address register, TLB physical address register 그리고 TLB miss address register가 있다. TLB miss address register는 instruction fetch와 Data access에 대해 각각 존재한다. [Table 5.1]는 TLB register에 대한 요약을 보여준다.

Table 5.1: TLB register's

TLB register name	Description
TLB index register	접근하고자 하는 TLB entry를 지정할 수 있다.
TLB virtual address register	가상 주소와 TLB control flag를 설정 한다.
TLB physical address register	물리 주소를 설정한다.
TLB process identifer register	Process ID를 설정한다. 이 레지스터를 접근하기 위해서는 general access index/data register를 사용한다.
Instruction TLB miss address register	Instruction fetch할 때 발생한 TLB miss의 가상주소를 저장한다. System coprocessor exception이 발생할 때 AE32000C는 자동적으로 fault가 발생한 pc값을 stack 영역에 저장되는데, instruction tlb miss address register를 사용하지 않고 이 pc값을 사용해도 된다. Pc 값을 사용할 것을 권장한다. 이 레지스터를 접근하기 위해서는 general access index/data register를 사용한다.
Data TLB miss address register	Data access 할 때 발생한 TLB miss의 가상 주소를 저장한다. 이 레지스터를 접근하기 위해서는 general access index/data register를 사용한다.

2) TLB Index Register

TLB index register는 접근하고자 하는 TLB entry를 지정한다.

Table 5.2: TLB Index Register (SCPR7)

SCPR7

Bit	R/W	Description	Default Value
31 : 9	RW	Reserved	-
8	RW	TLB select separated TLB인 경우 어떤 TLB에 접근할 것인지 나타낸다. 0 : select instruction TLB / unified TLB 1 : select data TLB	0b

Table 5.2: TLB Index Register (SCPR7)(계속)

Bit	R/W	Description	Default Value
7 : 3	RW	TLB index 선택된 TLB에서 어떤 entry를 선택할 것인지 결정한다. TLB의 각 way는 최대 32entry를 지닐 수 있으며, 이 값은 TLB virtual address[16:12]도 사용된다.	00h
2 : 1	RW	TLB way 선택된 TLB에서 지정된 entry가 어떤 way에 존재하는지 지정한다. 각 TLB마다 최대 4way를 가질 수 있다.	00b
0	RW	SCPR6 property SCPR6에 입력되는 주소의 형식을 지정한다. 0 : virtual address register 1 : physical address register	0b

TLB select 필드는 I-TLB와 D-TLB중 어느 것을 선택할지를 나타내며, TLB index는 32 개의 depth 중에서 어느 것을 선택할지를 나타낸다. TLB set은 4개의 way중에서 어느 것을 선택할지를 나타내며, TLB v/p address register select 는 선택된 entry에서 virtual address register를 선택할 것인지 physical address register를 선택할지를 나타낸다.
*AE32000C-Lucida*의 *TLB*는 하나의 *SRAM*을 통하여 *I-TLB*와 *D-TLB*를 구분하지 않는 *unified* 구조를 사용하고 있다.

3) TLB Virtual Address Register

Table 5.3: TLB Virtual Address Register (SCPR6)

SCPR6

Bit	R/W	Description	Default Value
31 : 17	RW	TLB virtual address[31:17] virtual page number의 상위 부분을 지닌다. 하위 부분은 SCPR7에서 지정된 TLB index 부분을 사용한다.	00000h
16	R	Reserved	0b
15 : 8	RW	Process ID 가상 메모리에서는 각 프로세스마다 동일 가상 주소를 지닐 수 있으므로, 이를 process ID를 이용하여 구분한다. TLB에서는 총 256개의 process를 구분할 수 있으며, OS에서는 사용하는 프로세스 번호가 이 이상으로 할당되는 경우에는 TLB miss handler에서 TLB entry를 place할 때 적절하게 할당해서 사용하여야 한다.	00h

Table 5.3: TLB Virtual Address Register (SCPR6)(계속)

Bit	R/W	Description	Default Value
7	RW	Global access 0 : access disable 1 : access enable	0b
6	RW	read only page 해당 page가 read only 모드인지 나타낸다. 해당 page에 대한 write 접근은 access violation을 발생시킨다. 0 : read only page 1 : read / write	0b
5	R	Page dirty 해당 페이지의 데이터가 변경된 경우 설정된다. 0 : page is clean 1 : page is dirty	0b
4 : 3	RW	cache configuration 00 : cache disable 01 : reserved 10 : cache enable with write-through 11 : cache enable with write-back	00b
2	RW	access right 0 : supervisor only 1 : supervisor / user	0b
1	RW	accessed 해당 페이지가 접근되었는지 나타낸다. TLB miss의 경우 replacement를 수행할 때 어떤 way를 replace시킬지 결정하는데 사용될 수 있다. 0 : page is not accessed 1 : page is accessed	0b
0	RW	TLB line valid 해당 라인이 valid한지 설정한다. 특정 TLB 라인에 대하여 invalidation을 수행하거나, 초기 설정에서 사용된다. 0 : invalid page 1 : valid page	0b

TLB virtual address register는 TLB에서 가상 주소에 대한 정보 TLB virtual address register는 SCPR7의 설정에서 0번째 bit가 '0'으로 설정되어 있어야 접근 가능하다. 이 레지스터는 다음과 같은 가상 주소 부분과 설정 정보를 포함하고 있다. 각각의 설정은 위의 표를 참고하도록 하자.

- 가상 주소 필드: 32 bit 주소 영역에서 page offset을 제외한 가상 주소
- 캐시 설정 필드: 해당 영역의 캐시 설정
- 접근 권한 필드: 해당 영역의 접근 권한(access right)
- Dirty page 필드: 해당 영역에 write가 수행된 적이 있으므로, 보조 기억 장치로의 이동시 write-back이 수행되어야 한다.
- Accessed 필드: 해당 페이지가 최근에 접근 되었음을 나타내며, 운영체제에서 TLB 교체를 수행할 때 어떤 way의 필드를 교체할 것인지 판단의 근거가 된다.
- Global Access 필드: 페이지 공유를 위하여 사용되며, 프로세스 ID와 무관하게 어떤 프로세스든지 접근할 수 있는 영역을 나타낸다.

4) TLB Physical Address Register

Table 5.4: TLB Physical Address Register (SCPR6)

SCPR6

Bit	R/W	Description	Default Value
31 : 12	RW	TLB physical address[31:12] physical page number를 설정하기 위하여 사용된다.	00000h
11 : 0	RW	Reserved	000h

TLB physical address register는 TLB physical address에 대한 정보를 포함하고 있다. physical address 필드는 가상 주소가 mapping 될 물리 주소를 나타낸다.

5) General Access Point Index Register

General access point(GAP)는 프로세서에서 사용하는 다양한 레지스터들을 설정하거나 모니터링 하기 위하여 사용되는 접근 포트이다.

TLB는 이 포트를 이용하여 miss가 발생한 주소 혹은 현재 구동중인 프로세스의 ID를 적어서 비교에 사용할 수 있다.

- GAP index 0x300: Instruction TLB miss address
- GAP index 0x301: Data TLB miss address
- GAP index 0x302: Process ID register

5.4 MMU Exceptions

이 절에서는 MMU 하드웨어 블록에서 발생시키는 exception에 대해 설명한다. MMU excetpion은 AE32000C의 메모리 접근 정보와 TLB entry의 정보가 일치하지 않을 때 발생한다. 크게 다음과 같이 나누어 볼 수 있다.

- TLB entry miss-match
  - Virtual address Miss : AE32000C에서 접근 하고자 하는 주소가 TLB entry에 존재하지 않을 때 발생함.
  - Process identifier Miss : application 프로그램에 할당된 process identifer가 TLB entry에 존재 하지 않을 때 발생함.
  - Invalid page : 프로그램에서 접근하려는 virtual address와 process identifer가 TLB entry에 존재하지만 TLB entry의 valid field가 off일 때 발생함.
- Page Protection
  - Write read only region : TLB entry hit인 경우에 read만 허가된 page 영역에 write동작이 일어날 때 발생함. (Instruction TLB에서 적용 안됨).
  - Access violation : TLB entry hit인 경우에 user mode에서 supervisor mode만 접근할 수 있는 메모리 영역을 접근할 때 발생함.

TLB entry miss–match와 Access violation 항목은 Instruction TLB와 Data TLB에 적용되며, write read only region항목은 Data TLB에만 적용되는 사항이다. 위와 같은 excetion인 상황에서 MMU는 AE32000C에게 TLB miss exception을 알린다. 그러면 AE32000C 는 현재 exception이 발생한 프로그램지점의 context를 저장한 후 cp0 exception hanlder를 호출한다. Cp0 exception handler에서는 exception이 상황이 무엇인지를 판별하여 적절한 처리를 한 후에 exception을 발생시켰던 프로그램으로 다시 복귀시킨다.

5.5 H/W user를 위한 TLB 구성

이 절에서는 TLB 하드웨어의 구성에 대해 설명한다. TLB는 4–way 128entry(각 way당 32entry)의 SRAM과 8개(instruction과 data)의 micro entry buffer를 가지는 2–Level 구조를 취하고 있다. 8개의 micro entry buffer는 SRAM의 일부 정보를 보관하여, 보다 빠른 address translation이 가능하도록 구성하였다.

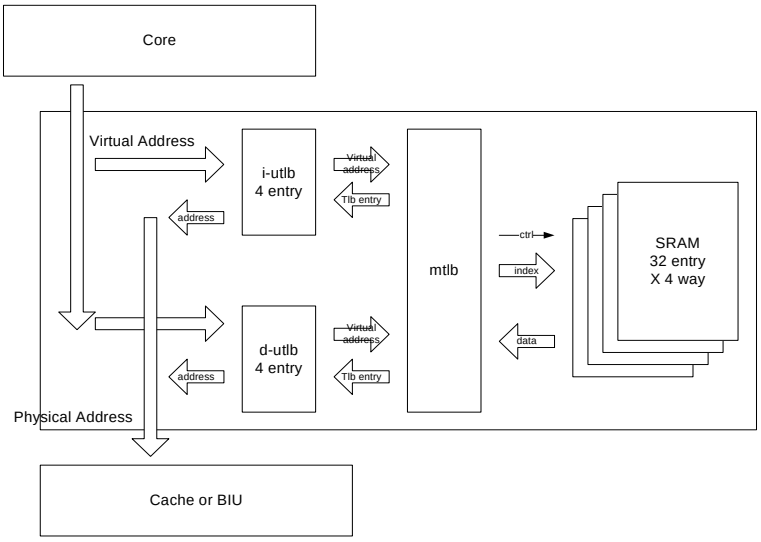


Figure 5.1: TLB Module TOP

TLB의 entry는 4개의 SRAM size에 의해 결정되는데, TLB SRAM의 수와 entry의 수를 조정하여 application의 환경에 맞게 조정이 가능하다. 조정이 가능한 범위는 다음과 같다.

Table 5.5: TLB 구성

SRAM	구조	Entry	Total Entry	비고
0		-	-	tlb 사용 안함
1	16x52(bit write) 32x52(bit write)	16 32	16 32	
2	16x52(bit write) 32x52(bit write)	16 32	32 64	
4	16x52(bit write) 32x52(bit write)	16 32	64 128	미지원

TLB SRAM의 수를 0으로 설정하는 것은 TLB를 사용하지 않는 것과 동일하다. 이외에 8개의 micro entry buffer의 조정이 가능하나, 이는 설계자(khlee@adc.co.kr)에 지원 요청을 하기 바란다.

5.6 TLB 설정 예제

이 절에서는 TLB register를 read/write하는 방법과 firmware level에서 TLB를 제어하는 순서를 어셈블리어와 함께 설명한다.

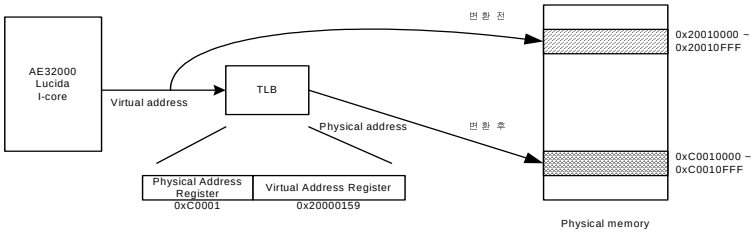


Figure 5.2: Address Translate example

그림은 Virtual address를 사용하여 0x20010000 – 0x20010FFF 영역에 접근시 TLB에 의하여 0xC0010000 – 0xC0010FFF 영역으로 변환되어 접근하는 상태를 나타낸다. 이러한 동작을 하기 위해서 TLB내에는 0x20010000 – 0x20010FFF 영역에 대한 변환 entry를 포함하고 있어야 한다. Seperated 구조의 TLB는 Instruction TLB와 Data TLB의 각각에 해당 영역에 대한 변환 entry를 설정하여야 한다.

5.6.1 TLB entry 설정

TLB entry를 설정하는 과정은 아래 예제와 같다.

```
1   # move %r0 to stack area for using %r0
2     push %r0
3
4
5   # Instruction fetch TLB setting
6
7   # TLB Index Register access
8   # TLB select : Instruction fetch TLB
9   # TLB index : 1
10  # TLB way : 1
11  # V/P Address Register Select : 0(Virtual)
12    ldi   0x0000000a, %r0
13    mvtc  0,          %r7
14
15
16  # TLB Virtual Address Reigster access
17  # TLB Virtual Address + spare bit : 0x2000
```

```
18  # Process Identifier : 0x01
19  # Read only page bit : 1
20  # The page is dirty bit : 0
21  # Cache Configuration : 11
22  # Access Right : 0
23  # Accessed : 0
24  # TLB line valid : 1
25    ldi   0x20000159, %r0
26    mvtc  0,          %r6
27
28
29  # TLB Index Register access
30  # TLB select : Instruction fetch TLB
31  # TLB index : 1
32  # TLB way : 1
33  # V/P Address Register Select : 1(Physical)
34    ldi   0x0000000b, %r0
35    mvtc  0,          %r7
36
37
38  # TLB Physical Address Reigster access
39  # TLB Physical Address : 0xC0001
40  # Reserved bit
41    ldi   0xC0001000, %r0
42    mvtc  0,          %r6
43
44
45
46  # Data TLB setting
47
48  # TLB Index Register access
49  # TLB select : Data TLB
50  # TLB index : 1
51  # TLB way : 1
52  # V/P Address Register Select : 0(Virtual)
53    ldi   0x0000010a, %r0
54    mvtc  0,          %r7
55
56
57  # TLB Virtual Address Reigster access
58  # TLB Virtual Address + spare bit : 0x2000
59  # Process Identifier : 0x01
60  # Read only page bit : 1
61  # The page is dirty bit : 0
```

```
62  # Cache Configuration : 11
63  # Access Right : 0
64  # Accessed : 0
65  # TLB line valid : 1
66    ldi   0x20000159, %r0
67    mvtc  0,          %r6
68
69
70  # TLB Index Register access
71  # TLB select : Data TLB
72  # TLB index : 1
73  # TLB way : 1
74  # V/P Address Register Select : 1(Physical)
75    ldi   0x0000010b, %r0
76    mvtc  0,          %r7
77
78
79  # TLB Physical Address Reigster access
80  # TLB Physical Address : 0xC0001
81  # Reserved bit
82    ldi   0xC0001000, %r0
83    mvtc  0,          %r6
84
85  # recover %r0, %r1 from stack areea
86    pop %r01
```

TLB entry를 설정시에 TLB index는 Virtual Address의 [17:12]와 동일하게 설정하여야 한다. 이는 TLB SRAM의 address로 사용되기 때문에 이 값이 다르게 설정되면 TLB matching 결과를 보장하지 못한다. TLB way는 Virtual Address 영역과 Physical Address 영역이 하나의 쌍을 이루도록 같은 값을 가져야 한다. V/P Address Register Select는 TLB entry의 Virtual Address 영역과 Physical Address 영역을 선택하며, 하나의 TLB entry를 설정하기 위해서 각 각의 영역을 선택하여 설정하여야 한다.
(주의사항)
위의 예제는 Virtual Address 영역을 설정하고 Physical Address 영역을 설정하는 순서를 보여주며, 이와 같은 순서를 따를것을 권장한다. 또한 Virtual Address Register는 메모리 설정을 함께 포함하고 있으며, 설정시에는 TLB line valid bit를 항상 1로 설정하여야 한다.

위와 같이 하나의 TLB entry를 설정하기 위해서는

1. mvtc %r7 (Virtual Address 선택)
2. mvtc %r6 (Virtual Address 설정)
3. mvtc %r7 (Physical Address 선택)

4. mvtc %r6 (Physical Address 설정)

의 과정을 거쳐야 한다.

5.6.2 TLB entry 설정 읽기

```
# move %r0 to stack area for using %r0
  push %r0


# Instruction fetch TLB setting

# TLB Index Register access
# TLB select : Instruction fetch TLB
# TLB index : 1
# TLB way : 1
# V/P Address Register Select : 0(Virtual)
  ldi   0x0000000a, %r0
  mvtc  0,          %r7


# TLB Virtual Address Reigster access
  mvfc  0,          %r6
  (type excute code)


# TLB Index Register access
# TLB select : Instruction fetch TLB
# TLB index : 1
# TLB way : 1
# V/P Address Register Select : 1(Physical)
  ldi   0x0000000b, %r0
  mvtc  0,          %r7


# TLB Physical Address Reigster access
  mvfc  0,          %r6
  (type excute code)



# Data TLB setting

# TLB Index Register access
```

```
39  # TLB select : Data TLB
40  # TLB index : 1
41  # TLB way : 1
42  # V/P Address Register Select : 0(Virtual)
43    ldi   0x0000010a, %r0
44    mvtc  0,          %r7
45
46
47  # TLB Virtual Address Reigster access
48    mvfc  0,          %r6
49    (type excute code)
50
51
52  # TLB Index Register access
53  # TLB select : Data TLB
54  # TLB index : 1
55  # TLB way : 1
56  # V/P Address Register Select : 1(Physical)
57    ldi   0x0000010b, %r0
58    mvtc  0,          %r7
59
60
61  # TLB Physical Address Reigster access
62    mvfc  0,          %r6
63    (type excute code)
64
65
66
67  # recover %r0 from stack areea
68    pop %r0
```

TLB entry를 읽을 시에는 Index Register를 통해 읽고자 하는 entry의 접근을 준비하고, mvfc 명령을 통하여 Virtual Address Register, 또는 Physical Address Register의 정보를 읽는다.

위와 같이 TLB entry의 설정을 읽고자 할때는 읽고자하는 Address 영역을 선택하고, mvfc 명령을 통해 정보를 GPR %r0에 읽어온다. GPR %r0는 임시 레지스터이므로 프로그래머는 이를 다른 레지스터에 옮겨 사용할것을 권장한다.

5.7 TLB를 사용하는 프로그램 흐름

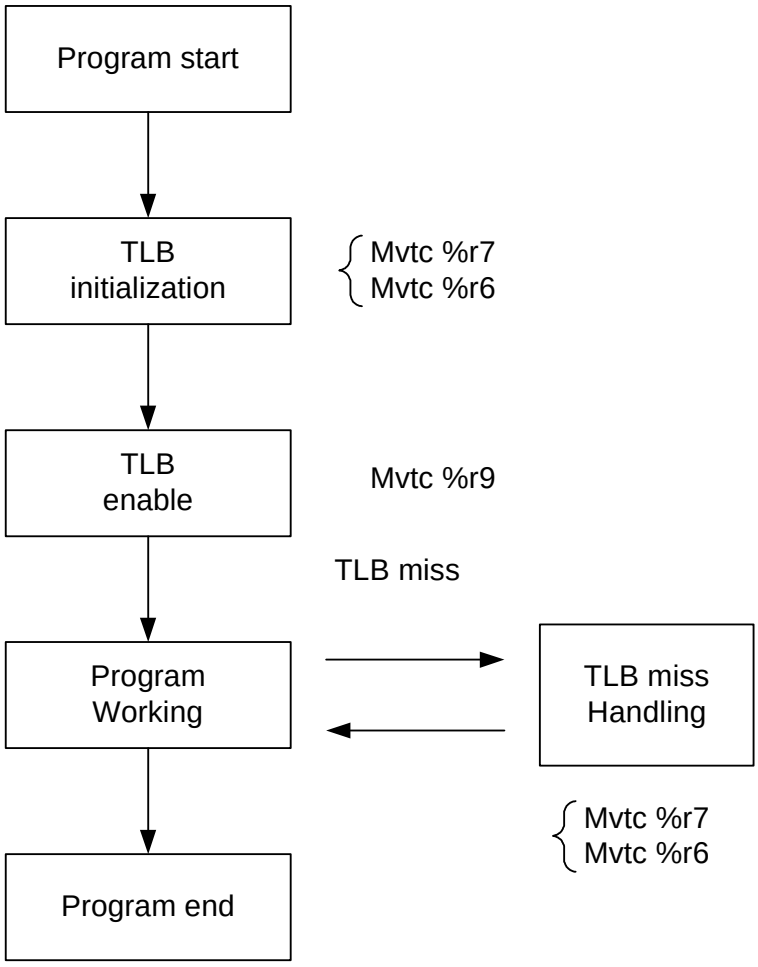


Figure 5.3: TLB Program Flow

[Figure 5.3]은 TLB를 사용할 때 프로그램 수행 순서도이다.

1. TLB initialization : TLB SRAM의 초기화. 그외에 TLB Invalidation, supervisor stack 영역이나 momory mapped I/O 영역을 TLB 변환 사상 영역으로 두기 위해 TLB entry에 정확한 정보를 미리 기입한다.
2. 뱅크 별로 TLB enable : TLB enable은 뱅크 별로 행하여 지기 때문에 뱅크의 enable에 신중을 기해야 한다.
3. main program 실행
4. TLB miss 발생 : TLB entry에 주소정보가 존재 하지 않을 때나, page table에 원하는 page 정보가 없거나, 원하는 데이터가 physcial memory에 존재하지 않을 때 발생한다.

5. Software적으로 TLB miss 처리한 후, fault가 발생한 지점으로 다시 복귀 : AE32000C-Lucida에 내장된 TLB는 소프트웨어적으로 관리하는 방식을 채택하고 있다. 따라서 TLB를 제어하는 운영 체제는 TLB miss 와 관련된 핸들러 작성에 세심한 주의를 기울려야 한다.

6. program이 종료 : program은 종료할 때 까지 3 5를 반복 수행할 수 있다.

(주의사항)
TLB를 사용하기 위해서 TLB SRAM의 초기화를 반드시 수행하여야 한다. TLB SRAM의 초기화시에 TLB의 예상치 못한 동작을 방지하기 위해 all'0 를 사용할것을 권장한다. 또한 TLB의 enable시에 TLB miss와 관련된 핸들러는 TLB enable영역에 두지 않도록 설정할것을 권장한다.

5.8 O/S 모델에 대한 예제

이 절에서는 다수의 프로그램이 동시에 수행되는 OS와 같은 환경에서 메모리의 접근을 안전하게 지원하기 위한 TLB의 접근 제어와 보호 기능에 대해 설명한다. OS와 같은 환경에서 kernel의 수행을 위해 필요한 메모리 영역은 다른 프로그램으로부터 접근을 제한하여야하며, 각각의 프로그램들 간에도 접근을 허용하지 않는 영역에 대한 설정이 필요하다. AE32000C-Lucida의 TLB는 Process ID(PID)와 Access right(접근권한)를 통해 허용되지 않은 영역의 접근을 제한한다.

다음은 다수의 프로그램이 동시에 수행되는 상황에서, TLB를 사용하여 프로그램의 수행에 따른 메모리관리 예제를 나타낸다.

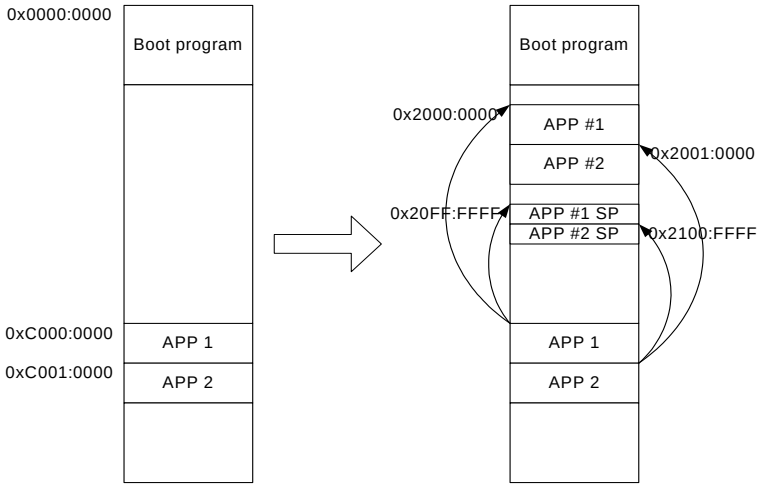


Figure 5.4: 다중 프로그램 수행시 TLB를 이용한 메모리 관리

Table 5.6: APP 구성

	APP 1	APP 2
시작주소	0x0000:0000	0x0000:0000
스택주소	0xC0FF:FFFF	0xC0FF:FFFF
TLB를 통한 시작주소 변경	0x0000:0000 –> 0x2000:0000	0x0000:0000 –> 0x2001:0000
TLB를 통한 스택주소 변경	0xC0FF:FFFF –> 0x20FF:FFFF	0xC0FF:FFFF –> 0x2100:FFFF
PID	0x01	0x02
Global access	0x0	0x0

Boot program은 시스템을 초기화하고 APP1과 APP2를 로딩하여 실행하는 프로그램

이다. APP1과 APP2는 동일한 시작주소와 스택주소를 갖는 프로그램이다. 이 프로그램을 TLB를 이용하지 않고 수행하게 되면 APP1과 APP2는 스택 영역의 데이터가 충돌하여 올바르게 수행되지 못한다.
TLB를 사용하는 경우에는 APP1의 시작주소와 스택주소를 표와 같이 변경하고, APP2의 시작주소와 스택주소를 표와 같이 변경한다. 이후 두 프로그램은 각기 다른 스택주소를 사용하여 진행한다.
TLB를 사용시에는 PID를 사용하여 의도하지 않은 접근을 방지할수 있다. APP1의 수행도중 예기치 않은 수행상의 오류로 APP1의 스택주소가 변경되어 APP2의 스택주소에 접근하게 되는 경우에는 PID가 일치하지 않기 때문에 CP0 exception이 발생한다. PID는 프로세스의 고유 번호를 나타내며, TLB entry를 설정하면서 지정한다. PID는 원칙적으로 중복을 배제하고 있지만, 255개의 최대 허용 수치를 넘는 프로그램의 수행시에는 적절히 중복하여 설정하여야 한다. 또한 여러 프로세스가 일정한 영역을 공유하여 사용하여야 하는 경우에는 Global access를 설정하여 사용할 수 있다. Global access가 설정되면 PID는 무시하고 접근이 이루어진다. 위의 예제에서는 프로그램의 실행 코드도 TLB를 통하여 관리하고 있다. 이를 위해 실행코드를 memory의 다른 영역에 옮기고 시작주소를 변경하였다.
TLB entry의 설정은 TLB 설정 예제를 참고하기 바란다.

Chapter 6

Buffer

이 장에서는 AE32000C-Lucida 프로세서와 시스템 버스간의 데이터 처리 효율을 향상시키기 위하여 사용되는 write buffer와 line fill buffer에 대하여 설명하고, 그 사용 방법에 대하여 살펴보도록 한다.

6.1 Introduction

AE32000C-Lucida 프로세서는 메모리 접근 과정에서 발생하는 Latency를 줄이고, 프로세서와 시스템 버스간의 데이터 처리 효율을 향상시키기 위하여 16-Byte 크기의 write buffer와 16-Byte 크기의 line fill buffer를 두고 있다.

6.2 Write Buffer

Write buffer는 프로세서가 메모리에 쓰기(write) 접근을 시도하는 경우에 메모리에 대한 쓰기 접근을 시도하는 대신 write buffer가 사용 가능한 경우 write buffer에 데이터를 쓰는 것만으로 쓰기 동작을 완료할 수 있도록 함으로써 쓰기 접근에 소모되는 시간을 감소시킬 수 있도록 만든다. Write buffer에 저장된 데이터는 시스템 버스에 별다른 동작이 없을 때 (혹은 write buffer가 꽉찬 경우) 메모리의 목적 주소에 쓰여진다. AE32000C-Lucida 프로세서는 16-Byte 크기의 write buffer를 채용하고 있으므로, 워드 단위(32-bit)의 데이터를 4개까지 저장할 수 있다. 따라서, 캐시가 활성화되지 않은 경우 혹은 캐시의 동작이 write-through 모드인 경우 뿐만 아니라, 캐시의 동작 모드가 write-back인 경우에도 하나의 캐시 라인을 저장하기 위하여 사용할 수 있으므로, AE32000C-Lucida에서는 write buffer와 더불어 6.3절에서 설명할 line fill buffer 버퍼를 같이 사용하여 write-back 캐시 동작시에 캐시 미스에 따른 라인 교체 과정에서 dirty 라인의 축출(eviction)에 소모되는 클럭을 효과적으로 줄여 성능 향상을 도모하는 concurrent line write-back 기법[1]을 채용하고 있다.

AE32000C-Lucida 프로세서는 write buffer의 동작 효율을 높이기 위하여 독립적으로 write된 작은 데이터들의 주소를 체크한 후 하나의 워드로 묶어서 처리할 수 있는 데이터를 한꺼번에 처리하는 byte gathering 기능을 지원한다. 또한, write buffer내에 존재하는 데이터를 프로세서에서 요구하는 경우 이를 직접 보내주는 data forwarding 기능을 지원한다. 만일 write buffer에 프로세서에서 요구하는 데이터의 일부만 존재하는 경우에는 메모리로부터 데이터를 읽어서 write buffer의 데이터와 조합함으로써, write buffer flushing 과정없이 항상 최신의 데이터가 프로세서로 전달될 수 있도록 지원하고 있다.

AE32000C-Lucida 프로세서의 write buffer는 접근하는 메모리 영역에 대하여 캐시가 활성화되어 있는 경우 항상 활성화되며, MBMU(Memory Bank Management Unit)을 통하여 결정할 수 있다.

[1] 이 기법은 background write-back 혹은 fly-by write-back으로도 불린다.

6.3 Line Fill Buffer

Line fill buffer는 캐시의 라인 교체(line replace) 동작을 효과적으로 수행하기 위하여, 캐시로 부터 축출(eviction)될 dirty line이 write buffer로 쓰여지는 동안 메모리로부터 캐시 라인을 채울 데이터를 읽어서 저장하고 있는데 사용하는 버퍼이다. 이러한 동작을 통하여 그림. 6.1에서 보여지는 바와 같이 캐시 라인의 교체 과정에서 발생하는 write-back에 따른 지연을 효과적으로 감출 수 있으므로, 이를 concurrent line write-back 기법이라 부른다. Line fill buffer는 캐시의 동작 모드가 write back 모드로 설정된 경우에만 활성화되며, write buffer와 조합하여 사용된다.

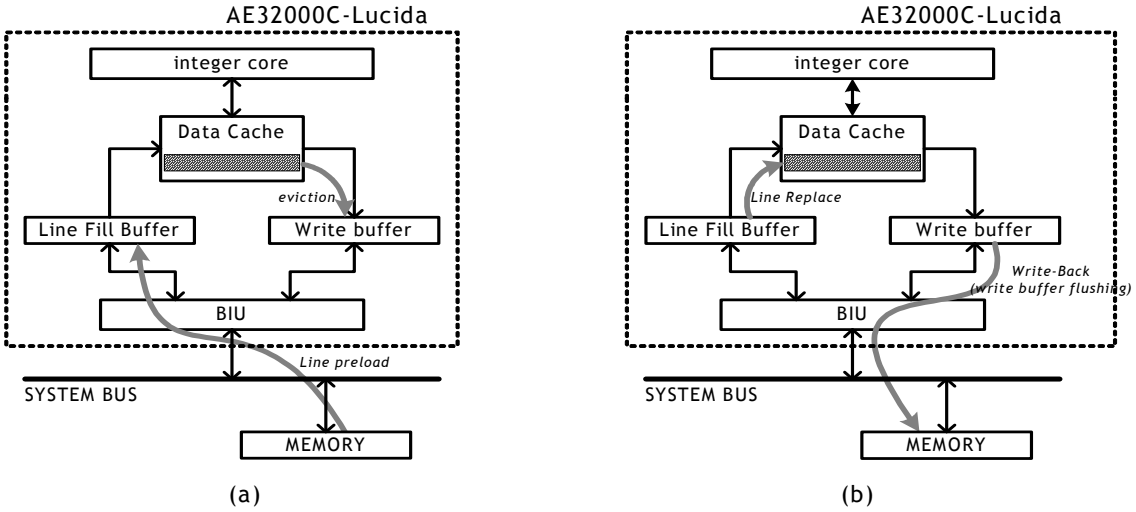


Figure 6.1: Concurrent line write-back procedure; line fill buffer reads line data meanwhile a dirty line is evicted to the write buffer.

Chapter 7

SPM

이 장에서는 캐시와 동일한 수준의 접근 시간을 보장하는 내장 메모리인 Scratch-PAD Memory(SPM)를 AE32000C-Lucida 프로세서에서 적용하여 사용하는 방법에 대하여 설명하도록 한다.

7.1 SPM Overview

AE32000C-Lucida 프로세서는 메모리 접근에 걸리는 Latency를 감소시키기 위해 Scratch-PAD Memory(SPM)를 사용한다. SPM은 어플리케이션을 제어하는데 정확한 응답성을 제공하는 메모리로 ARM 프로세서의 TCM(Tightly Coupled Memory)과 유사하다. 일반적으로 SPM은 성능에 중요한 코드, 정확한 접근 시간 및 빠른 처리가 필요한 코드, 데이터를 저장하는데 사용된다.

SPM은 software managed cache와 같이 소프트웨어에 의하여 관리된다. 자주 사용되는 데이터/명령어를 SPM에 적재함으로써 real-time 시스템과 같은 예측 가능한 응답을 원하는 어플리케이션에 캐시보다 정확한 응답을 보장 할 수 있다. SPM은 일반적으로 다음과 같은 경우에 효율적으로 사용되어진다.

- Real-Time과 같은 예측 가능한 응답이 요구되는 어플리케이션
- 빠른 처리가 필요한 명령어 영역
  - 빠른 처리 및 예측 가능한 응답이 필요한 인터럽트
  - 자주 사용되어지며, 빠른 처리가 필요한 함수 및 루프
- 빠른 처리가 필요한 데이터 영역
- 인터럽트 처리를 위한 Exception Stack 저장

7.1.1 Feature

AE32000C-Lucida 프로세서에 적용된 SPM은 다음과 같은 특징을 가진다.

- Harvard 구조의 명령어/데이터 SPM
- Integer Core의 1cycle 명령어/데이터 접근 보장 [1]
- SPM 영역 설정 가능 : 사용자 원하는 메모리 영역으로 SPM 설정 가능
- 외부 Master에서 SPM 접근 가능 (Implementation 단계에서 결정)

[1] 명령어 버스에서 데이터 SPM 접근 및 데이터 버스에서 명령어 SPM 접근시에는 1cycle이 보장되지 않는다.

7.2 SPM Type

SPM은 응용에서의 요구 및 하드웨어 비용에 따라 다양한 크기 및 형태로 구성될 수 있다. 본 절에서는 SPM의 타입별 구성에 대해 설명한다.

7.2.1 SPM 타입별 구성 형태

AE32000C-Lucida 프로세서에 적용된 SPM은 2가지 형태로 구성할 수 있으며, 그림 7.1은 SPM의 구성 형태를 나타내고 있다[2].

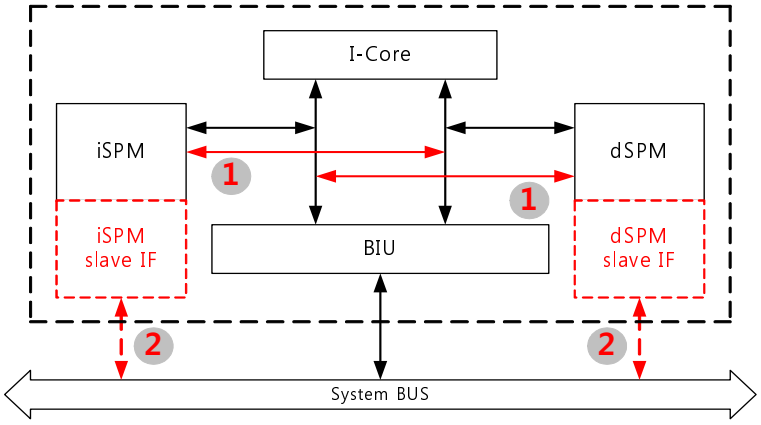


Figure 7.1: SPM Type 구성 형태

- TYPE 1
  - 정수 코어의 명령어 버스에서 데이터 SPM에 접근 할 수 있다.
  - 정수 코어를 이용해서만 SPM에 명령어/데이터를 적재할 수 있다.
- TYPE 2 : Implementation 단계에서 추가할 수 있음
  - TYPE 1과 같이 정수 코어의 명령어 버스에서 데이터 SPM에 접근 할 수 있다.
  - 정수 코어를 이용해서 SPM에 명령어/데이터를 적재할 수 있다.
  - SPM에서 slave 인터페이스를 지원하기 때문에 정수 코어가 아닌 DMA와 같은 다른 마스터에서 SPM에 데이터를 적재 할 수 있다.
  - 현재는 AHB slave 인터페이스만 지원한다.
  - TYPE1에서 그림7.1의 2번이 추가된 구성이다.

7.2.2 SPM 타입별 장단점

표7.1은 AE32000C-Lucida 프로세서에 적용된 SPM의 타입의 장단점을 나타낸 것으로 SPM의 사용 목적에 따라 뚜렷히 구별 될 수 있는 것을 알 수 있다.

[2]SPM의 형태는 implementation 단계에서 결정이 된다. 해당 내용은 **??**절을 참고하도록 한다.

Table 7.1: 각 타입별 장단점

타입	내용	
장 점	TYPE 1	외부 Master에서 SPM으로 접근이 없는 경우 모든 응용 프로그램에서 SPM을 이용할 수 있다.
	TYPE 2	외부 Master에서도 SPM의 접근이 가능한 가장 강력한 기능을 지원하는 형태이다.
단 점	TYPE 1	외부의 Master에서 SPM의 접근이 불가능 하다.
	TYPE 2	가장 자유롭게 SPM에 접근 할 수 있지만 가장 많은 하드웨어가 필요하다.

- 펌웨어 수준의 마이크로 컨트롤러에서는 외부 마스터에 의한 SPM을 접근하는 어플리케이션이 사용되지 않기 때문에 TYPE 1의 형태의 SPM이 적당할 것이다. 또한 OS 형태의 데이타 영역에 응용 프로그램이 구동되는 어플리케이션에서도 적용할 수 있다.
- TYPE 2 형태의 SPM은 외부 Master에서 접근 할 수 있는 강력한 기능을 제공하지만 SPM이 프로세서와 밀접하게 동작을 하기 위한 메모리이기 때문에 외부 Master의 접근이 얼마나 효율적으로 사용될지 현재 상황에서는 의문이다 [3]. MPSoC 같은 형태의 시스템에서 SPM을 다양한 방법으로 제어 및 사용 할 수 있을 것이다.

[3] DMA를 이용하여 SPM에 명령어/데이타를 적재하는 경우에는 효과적이다.

7.3 SPM 아키텍쳐

AE32000C-Lucida 프로세서에 적용된 SPM은 내부 레지스터 설정에 따라 사용자가 임의대로 내부를 재구성하여 사용할 수 있다. 즉, Implementation 단계에 결정된 physical SRAM의 개수를 이용하여, 사용자는 자신의 프로그램에 최적화된 SPM 을 구성할 수 있다. 또한 각 레지스터들은 GAP(General Access Pointer)를 이용하여 제어한다 [4].

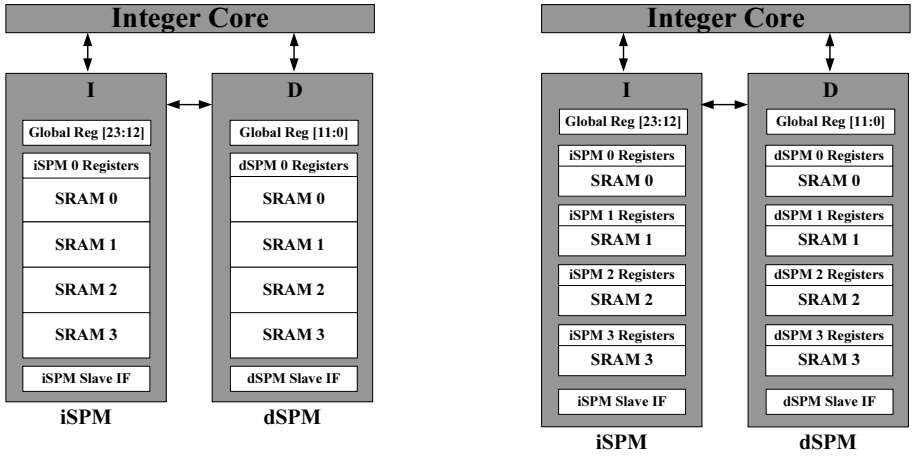


Figure 7.2: Examples of Reconfigurable SPM

- 사용자 입장에서는 명령어/데이터 SPM을 구분하여 사용.
- 명령어/데이터 SPM 각각은 여러 개의 메모리 덩어리(N-KB/Bank)로 구성될 수 있음
- Virtual Addressing 만 지원. (TLB ON 영역에 대해서는 Virtual address와 Phisical address가 1:1 Mapping 됨을 보장해야 함)
- SPM의 모든 register는 MVTC, MVFC를 사용. (관리자 모드에서만 설정 가능)
- SPM은 1개의 Golbal Register와 N개의 Local Register들을 가짐.
- 메모리 일관성 문제(SPM/cache)는 보장하지 않음.
- SPM 데이터 이동 방법
  - 접근 권한이 주어진 영역에대해 프로세서가 LD/ST 명령어로써 데이터 이동 가능.
  - DMA등 다른 Master에 의한 데이터 이동 가능하나 접근 권한은 프로세서의 모드를 참조하므로 프로세서의 모드에 주의하여서 사용하도록 함.
  - 권한 설정이 잘못된 경우에는 Access violation이 발생.

[4] 각각의 레지스터들은 spm_param.vh 에서 초기값을 설정 할 수 있다.

7.4 SPM Registers

SPM 전체를 관장하는 1개의 Global Control Register를 갖는다. 또한 SPM은 내부에 여러 개의 Bank로 구성될 수 있으므로 Global Register의 Configuration에 의해 결정되는 Bank 개수 만큼 Local Register Set을 갖는다. Local Register Set는 다음과 같은 3개의 32bit Register로 구성된다.

- Local SPM Control Register
- Local SPM Start Address
- Local SPM End Address [5]

7.4.1 Global SPM Control Register

Table 7.2: SPM Control Register

ADDRESS : 0x700 -SPM Control Reg.

Bit	R/W	Description	Default
31 : 28	R	**Exception Status** 4'b0001 : DATA Access Violation 4'b0010 : Instruction Access Violation	0h
27 : 24	R	Reserved	0h
23 : 20	R	**iBank Size: iSPM**에서 각 **bank**의 **physical Memory** 크기 4'h0 : 1 KB 4'h1 : 2 KB 4'h2 : 4 KB 4'h3 : 8 KB 4'h4 : 16 KB 4'h5 : 32 KB 4'h6 : 64 KB 4'h7 : 128 KB 4'h8 : 256 KB	
19 : 16	R/W	**iSPM Configuration** 4'h0 : 사용자에게 1개의 메모리 덩어리로 보임 4'h1 : Reserved 4'h2 : 사용자에게 4개의 메모리 덩어리로 보임 (4개 를 넘는 경우는 현재 구현되어 있지 않음)	0h

[5] 하드웨어적으로 SPM 영역은 시작과 끝주소를 포함하기 때문에 이에 유의하여 사용하도록 한다.

Bit	R/W	Description	Default
15 : 12	R	**iSPM Enable** 4'b0001 : SPM Enable 4'b0000 : SPM Disable	0h
11 : 8	R	**dBank Size: dSPM**에서 각 **bank**의 **physical Memory** 크기 4'h0 : 1 KB 4'h1 : 2 KB 4'h2 : 4 KB 4'h3 : 8 KB 4'h4 : 16 KB 4'h5 : 32 KB 4'h6 : 64 KB 4'h7 : 128 KB 4'h8 : 256 KB	
7 : 4	R/W	**dSPM Configuration** 4'h0 : 사용자에게 1개의 메모리 덩어리로 보임 4'h1 : Reserved 4'h2 : 사용자에게 4개의 메모리 덩어리로 보임 (4개 를 넘는 경우는 현재 구현되어 있지 않음)	0h
3 : 0	R	**dSPM Enable** 4'b0001 : SPM Enable 4'b0000 : SPM Disable	0h

7.4.2 Local SPM Control Register

Table 7.3: Local SPM Control Register

ADDRESS : 0x701, 0x711, 0x721, 0x721 - Local iSPM Control Reg.
ADDRESS : 0x704, 0x714, 0x724, 0x724 - Local dSPM Control Reg.

Bit	R/W	Description	Default
31 : 12	R	Reserved	0h
11 : 8	R	**External Access: BUS** 접근 권한 4'h0 : External Access Not Support 4'h1 : External Access Support	
7 : 4	R/W	**Priviedge Mode:** 사용자 권한 4'h0 : Supervisor only Access 4'h1 : Supervisor/User Access	0h

Bit	R/W	Description	Default
3 : 0	R	**Enable** 4'b0001 : Local SPM Enable 4'b0000 : Local SPM Disable	0h

7.4.3 Local SPM Start Address Register

Table 7.4: SPM Start Address Register

ADDRESS : 0x702, 0x712, 0x722, 0x732 - Local iSPM Start Register
ADDRESS : 0x705, 0x715, 0x725, 0x735 - Local dSPM Start Register

Bit	R/W	Description	Default
31 : 0	R/W	**SPM Start Address**	0h

7.4.4 Local SPM End Address Register

Table 7.5: SPM End Address Register

ADDRESS : 0x703, 0x713, 0x723, 0x733 - Local iSPM End Register
ADDRESS : 0x703, 0x713, 0x723, 0x733 - Local dSPM End Register

Bit	R/W	Description	Default
31 : 0	R/W	**SPM End Address**	0h


End Address Register의 경우 만드시 Start Address Register를 먼저 설정한 뒤 설정 하도록 한다.

7.4.5 SPM 설정 예제

다음은 Implementation 상에서 iSPM/dSPM 각각 4개의 4KB SRAM을 사용한 SPM Controller 예제이다. 각각은 레지스터 설정을 통해 1개, 4개의 Logical Memory로 사용하는 예를 보여주고 있으며

본 절에서는 그림7.3 중 4개의 Logical Memory를 설정하고 이를 활용하는 예제에 대한 코드에 대해 설명한다. SPM 레지스터의 설정은 GAP를 이용하기 때문에 co-processor 레지스터 접근 명령어인 MVTC와 MVFC를 사용하게 된다. 두개의 영역을 SPM에 설정하는 예는 다음과 같다.

- Physical Memory 개수 : 8 개 (iSPM/dSPM 각각 4개)
- Physical Memory Size : 32 KB (iSPM/dSPM 각각 16KB)
- Logical Memory Bank 개수 : 7 개 (iSPM/dSPM 각각 3개, 4개)

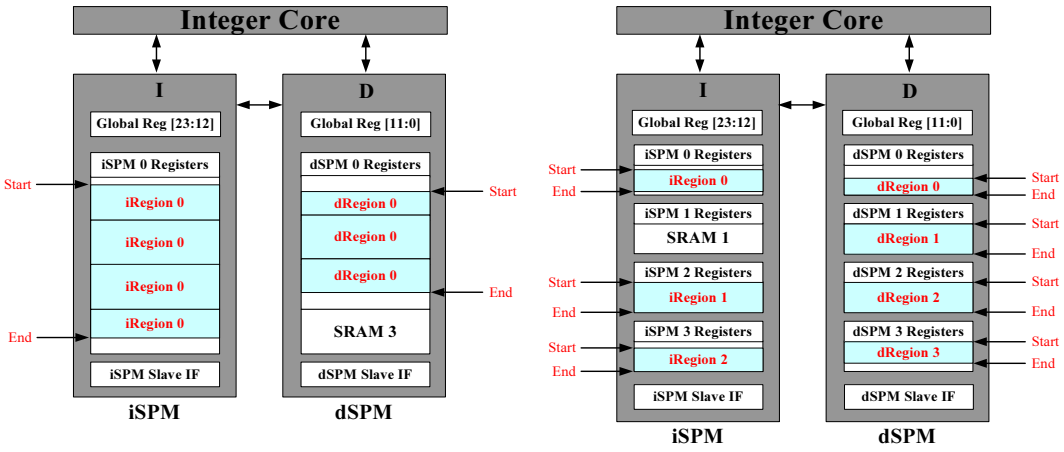


Figure 7.3: Configuration Examples

- Logical Memory 영역 :

  iRegion 0 : 0x0000_0100 ~ 0x0000_0C00

  iRegion 1 : 0x0000_1000 ~ 0x0000_1FFF

  iRegion 2 : 0x0000_3200 ~ 0x0000_3FFF

  dRegion 0 : 0xC000_0800 ~ 0xC000_0FFF

  dRegion 1 : 0xC000_1000 ~ 0xC000_1FFF

  dRegion 2 : 0xC002_8000 ~ 0xC002_8FFF

  dRegion 3 : 0xC002_A000 ~ 0xC002_AE00

```
1
2   /* Macro function */
3   void gap_write (int index, int value) {
4     asm("  lea  (%0, 0), %%r0 " : : "r" (index) : "r0" );
5     asm("  mvtc 0, %r3       ");
6     asm("  lea  (%0, 0), %%r0 " : : "r" (value) : "r0" );
7     asm("  mvtc 0, %r4       ");
8   }
9
10
11  /* SPM Global Register Setting */
12    gap_write(0x700, 0x00021021);   // #on #num of memory bank: 4
13
14  /* Copy Routines */
15    // 방법 1. LD, ST 를 사용하여 프로세서가 직접 SPM에 데이터를 채움
16              (이 경우 각 SPM Start/End Regiser를 원하는 임의 주소로 설정한 다음
17              사용해야 하며 각 Bank를 Enable 해야함)
18    // 방법 2. LD, ST 를 사용하여 BUS를 통해 프로세서가 SPM에 데이터를 채움
```

```
19              (이 경우 SPM의 주소는 해당 플랫폼의 Memory Map을 참고할 것)
20    // 방법 3. DMA 등 프로세서 이외의 BUS Master를 사용하여 SPM에 데이터를 채움
21              (이 경우 SPM의 주소는 해당 플랫폼의 Memory Map을 참고할 것)
22
23  /* SPM Local Register Setting */
24    /* Setting iRegion 0. -> Physical MEM 0. */
25    gap_write(0x702, 0x0100);       // Setting Logical Bank0 START address : 0x0100
26    gap_write(0x703, 0x0C00);       // Setting Logical Bank0 END   address : 0x0C00
27    gap_write(0x701, 0x1);          // Setting Logical Bank0 On
28
29    /* Setting iRegion 1. -> Physical MEM 2. */
30    gap_write(0x722, 0x1000);       // Setting Logical Bank1 START address : 0x1000
31    gap_write(0x723, 0x1FFF);       // Setting Logical Bank1 END   address : 0x1FFF
32    gap_write(0x721, 0x1);          // Setting Logical Bank1 On
33
34    /* Setting iRegion 2. -> Physical MEM 3. */
35    gap_write(0x732, 0x3200);       // Setting Logical Bank2 START address : 0x3200
36    gap_write(0x733, 0x3FFF);       // Setting Logical Bank2 END   address : 0x3FFF
37    gap_write(0x731, 0x1);          // Setting Logical Bank2 On
38
39
40    /* Setting dRegion 0. -> Physical MEM 0. */
41    gap_write(0x705, 0xC0000800);   // Setting Logical Bank0 START address : 0xC000_0800
42    gap_write(0x706, 0xC0000FFF);   // Setting Logical Bank0 END   address : 0xC000_0FFF
43    gap_write(0x704, 0x1);          // Setting Logical Bank0 On
44
45    /* Setting dRegion 1. -> Physical MEM 1. */
46    gap_write(0x715, 0xC0001000);   // Setting Logical Bank1 START address : 0xC000_1000
47    gap_write(0x716, 0xC0001FFF);   // Setting Logical Bank1 END   address : 0xC000_1FFF
48    gap_write(0x714, 0x1);          // Setting Logical Bank1 On
49
50    /* Setting dRegion 2. -> Physical MEM 2. */
51    gap_write(0x725, 0xC0028000);   // Setting Logical Bank2 START address : 0xC002_8000
52    gap_write(0x726, 0xC0028FFF);   // Setting Logical Bank2 END   address : 0xC002_8FFF
53    gap_write(0x724, 0x1);          // Setting Logical Bank2 On
54
55    /* Setting dRegion 3. -> Physical MEM 3. */
56    gap_write(0x735, 0xC002A000);   // Setting Logical Bank3 START address : 0xC002_A000
57    gap_write(0x736, 0xC002AE00);   // Setting Logical Bank3 END   address : 0xC002_AE00
58    gap_write(0x734, 0x1);          // Setting Logical Bank3 On
59
```

Chapter 8

AMBA AHB and AXI

이 장에서는 AE32000C-Lucida 프로세서의 버스 인터페이스에 대하여 설명하도록 한다.

8.1 Bus Interface

AE32000C-Lucida 프로세서는 시스템 버스로서 여러 SoC에서 폭넓게 받아들여지고 있는 ARM 사의 AMBA Bus 프로토콜을 사용하고 있다. 이는 기존의 AMBA버스 기반으로 제작되어 있는 많은 주변 장치와 AE32000C-Lucida 프로세서를 이용하여 쉽게 SoC를 만들 수 있음을 의미한다.

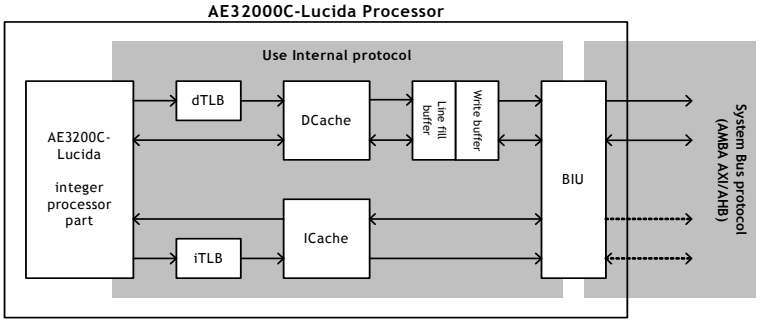


Figure 8.1: AE32000C-Lucida Processor BIU

AE32000C-Lucida 프로세서의 버스 계층 구조: 정수프로세서 수준에서 분리된 버스를 지니고, 이는 L1 캐시와 연결된다. 캐시는 버스인터페이스를 통하여 시스템 버스와 연결되는 형태를 취하고 있다.
BIU는 프로세서 내부 인터페이스을 시스템 버스의 프로토콜로 변환하는 형태를 사용하고 있다.

AE32000C-Lucida의 BIU는 Instruction path와 Data path에 대하여 각각 독립적으로 구성이 가능하다. 이는 프로세서의 내부가 Instruction path와 Data path가 분리된 하버드 아키텍쳐를 지니고 있기 때문이다. 이러한 이유로 시스템 버스도 Instruction 버스와 Data 버스로 분리하여 구성할수 있으며, 분리된 시스템 버스는 Instruction과 Data간의 버스 점유에 따른 지연이 없기 때문에 전체 시스템 성능을 향상시킬수 있다. 그러나 분리된 시스템 버스는 버스를 이중으로 구성해야 하는 오버헤드가 따른다.

8.2 지원 버스 프로토콜

BIU 는 다음의 프로토콜을 지원하며, 프로토콜의 선택은 SoC의 특성에 따라 구성하여 사용할수 있다.

- AHB protocol
  AE32000C-Lucida 프로세서는 일반적으로 많이 사용되고 있는 AHB protocol를 사용하여 다양한 활용성을 가지고 있다. AE32000C-Lucida 프로세서의 BIU는 AHB protocol을 내부 자원의 활용도에 적합하도록 일부 기능만을 지원하고 있다.
  AE32000C-Lucida 프로세서의 기본 동작은 1 word(또는 Short/byte)를 읽고 쓰는 동작이며, 캐시의 동작중에는 한번에 4word를 읽고 쓰기가 가능하다.
  AHB protocol은 burst transfer를 지원하며, 다양한 burst mode를 지니고 있는데, 이중 AE32000C-Lucida 프로세서의 BIU는 4 word Incremental mode를 지원한다.
- AXI protocol
  AE32000C-Lucida 프로세서는 AXI protocol을 지원한다. AXI protocol은 ARM 사의 bus protocol로 AHB보다 높은 성능을 요구하는 SoC환경을 위한 protocol이다.
  AE32000C-Lucida의 BIU는 AXI protocol을 통해 AHB와 동일한 기능을 수행하며, 각각 독립된 instruction path와 data path를 지원한다.(공유하는 것도 가능하다.)
  AXI protocol을 사용함으로써 BUS의 대역폭으로 인한 성능의 저하를 크게 개선시킬수 있다.

8.3 BIU의 입력과 출력 신호

AMBA bus 시스템에는 능동적으로 memory의 읽기와 쓰기 동작을 주관하고, 주소와 컨트롤을 제공하는 MASTER와 해당 주소 영역의 읽기와 쓰기를 수행하며, MASTER의 동작에 응답을 하는 SLAVE가 존재하며, MASTER와 SLAVE를 연결하고 BUS(BUS는 주소와 컨트롤, 그리고 data가 이동하는 통로이다.)를 관리하는 ARBITER가 존재한다.
AHB와 AXI 모두 이러한 구성 형태를 가지며, 각 protocol에 따라 입 출력 시그널과 동작 방식에 차이를 보인다.

(본 문서는 이해를 돕기 위해 일부 시그널만 표시한다. 자세한 사항은 AMBA spec 문서를 참고하기를 바란다)

8.3.1 AHB 인터페이스

AHB 표준 인터페이스는 다음과 같은 입 출력 시그널로 구성된다.

- HADDR : 버스를 통하여 접근하고자 하는 주소를 나타낸다. 32bit width를 가진다.
- HWRITE : 읽기와 쓰기를 구분한다.
- HBURST : Burst mode를 나타낸다. AE32000C-Lucida의 BIU는 Incremental 모드를 지원한다.
- HTRANSE : 현재 transfer의 상태를 나타낸다.
- HSIZE : 현재 transfer의 데이터의 크기를 나타낸다. AE32000C-Lucida의 BIU는 8bit, 16bit, 32bit을 지원한다.
- HREADY : transfer의 동작이 완료되었음을 나타낸다.
- HRDATA : 버스를 통한 읽기 동작의 데이터를 나타낸다. 32bit width를 가진다.
- HWDATA : 버스를 통한 쓰기 동작의 데이터를 나타낸다. 32bit width를 가진다.

8.3.2 AXI 인터페이스

AXI 표준 인터페이스는 5개의 채널로 구성되며, 각 채널은 다음과 같다.

- Read Address 채널 : 읽기 주소와 제어 시그널을 전송한다.
- Read Data 채널 : SLAVE의 읽기 동작에 대한 응답과 memory의 data를 전송 받는다.
- Write Address 채널 : 쓰기 주소와 제어 시그널을 전송한다.
- Write Data 채널 : SLAVE를 통해 memory에 쓰고자하는 data를 전송한다.
- Write Response 채널 : 쓰기 동작에 대한 응답을 전송받는다.

입출력의 주소와 데이터의 width는 AHB와 동일하게 32bit로 구성된다. 각 채널의 입출력 시그널은 AMBA AXI spec을 참고하기 바란다.

8.3.3 예외 상황의 처리

AE32000C-Lucida 프로세서에서 발생할수 있는 예외 상황으로는 인터럽트의 발생과 프로세서의 상태 변화에 따른 제어가 있다.

- 인터럽트
  AE32000C-Lucida 프로세서는 소프트웨어 인터럽트와 하드웨어 인터럽트를 지원한다.
  소프트웨어 인터럽트는 관리자의 자원을 사용자 모드에서 접근하고자 호출하며, 지정된 인터럽트 번호로 프로그램의 흐름을 변경하여 진행한다.
  하드웨어 인터럽트는 외부 하드웨어에 의하여 발생하며, 인터럽트 컨트롤러를 통해 인터럽트의 종류를 파악하여 해당 인터럽트 번호로 프로그램의 흐름을 변경하여 진행한다. 하드웨어 인터럽트가 발생하면 AE32000C-Lucida 프로세서는 버스의 인터럽트 컨트롤러에 접근하여 인터럽트 번호를 읽어온다. 이때 AHB 또는 AXI protocol을 이용하여 SLAVE로 존재하는 인터럽트 컨트롤러에 접근하게 된다.

- special cycle
  프로스서의 상태 변화를 외부에 알려야 할 필요가 있는 상태를 나타낸다.
  인터럽트가 발생하여 인터럽트 컨트롤러에 인터럽트의 처리가능상태를 알려줘야 하는 경우, 프로세서의 동작을 진행하지 못하는 에러가 발생하여 PMU(Power Management Unit)를 제어하고자 할때, 또는 전력관리를 위하여 PMU에 접근을 시도하는 경우들이 포함된다. 이때 AHB 또는 AXI protocol의 쓰기 동작을 통하여 SoC의 인터럽트 컨트롤러 또는 PMU에 상태를 전송한다.

AE32000C-Lucida 프로세서는 인터럽트 또는 special cycle를 처리하기 위해서 해당 인터럽트 컨트롤러의 주소와 PMU의 주소의 접근이 가능하여야 한다. 이러한 주소의 정보는 AE32000C-Lucida 프로세서의 BIU에 하드웨어로 정의되어있기 때문에, SoC의 구축시에는 이에 대한 변경이 필요하다. (주의 사항) SoC 구축을 위한 프로토콜의 변경이나 주소 정보의 변경시에는 담당자에게 문의하기 바란다.

Appendix A

System Coprocessor Register

A.1 System Coprocessor Register

Number	R/W	Description	Remark
SCPR0	R	Reserved	
SCPR1	R	Reserved	
SCPR2	R	Reserved	
SCPR3	RW	General Access Point Index Register	GAP
SCPR4	RW	General Access Point Data Register	GAP
SCPR5	RW	Sub-bank Address Register	
SCPR6	RW	TLB Virtual Address Reigster	
	RW	TLB Physical Address Register	
SCPR7	RW	TLB Index Register	
SCPR8	RW	Sub-bank Index / Configuration Register	
SCPR9	RW	Memory Bank Register	
SCPR10	R	Reserved	
SCPR11	W	Cache Control Register	
SCPR12	RW	Vector Base Register	
SCPR13	RW	User Stack Pointer Register	
SCPR14	R	Reserved	
SCPR15	W	Master Command Register	
	R	Status Register	

A.1.1 Status Register

SCPR15 (read only)

Bit	R/W	Description	Default Value
31	R	Access Right (Privilege) 보조 프로세서의 접근 권한을 나타낸다. 시스템 보조 프로세서는 항상 Supervisor only 이므로 '1'로 설정된다. 0 : supervisor / user accessable 1 : supervisor access only (default and only)	1b
30:28	R	Type Number 보조 프로세서의 형태 번호	001b
27:25	R	Subtype Number 보조 프로세서의 부형태 번호	000b
24:23	R	Reserved	00b
22:21	R	Reserved	00b

Bit	R/W	Description	Default Value
20:19	R	TLB configuration 프로세서에서 제공되는 TLB의 형태를 나타낸다. 00 : No TLB 01 : unified 4-way, 128 entry 10 : separated 4-way, 64 entries for each 11 : separated 4-way, 128 entries for each	00b
18	R	L1 Cache presented 프로세서에서 L1 cache가 제공되는지 나타낸다. 0 : L1 Cache is presented 1 : L1 Cache is not presented	0b
17	R	L1 Cache Snooping Capability 프로세서에서 제공되는 L1 cache가 snooping 기능을 수행하는지 나타낸다. L1 cache가 제공될 때만 유효하다. 0 : L1 cache supports snooping 1 : L1 cache doesn't supports snooping	1b
16	R	L1 Cache Replacement Policy 프로세서에서 제공되는 L1 cache에서 제공되는 replacement policy를 나타낸다. L1 cache가 제공될 때만 유효하다. 0 : L1 cache supports write-through only 1 : L1 cache supports write-through and write-back	1b
15:10	R	Reserved	00h
9:8	R	Reserved	00b
7	R	Reserved	0b
6	R	Misalign Correction Support 현재 장착된 SCP가 하드웨어적인 misalign correction을 지원하는지 나타낸다. Misalign correction 기능은 data 접근시에만 적용된다. 0 : Misalign Correction is not supported 1 : Misalign Correction is supported	0b

Bit	R/W	Description	Default Value
5:2	R	SCP Pending Exception Number SCP exception이 발생한 경우 발생한 exception의 종류를 저장하고 있는 부분이다. Exception 처리가 종결될 경우 MCR의 end of exception 부분을 이용하여 예외처리가 종결되었음을 알려야 한다. 0000 : instruction access violation 0001 : instruction TLB miss 0010 : privilege violation exception 0011 : data address misalignment 0100 : data access violation 0101 : data TLB miss 0110 : data write to read-only region 1000 : instruction address misalignment 1111 : N / A	1111b
1	R	Status of SCP Pending Exception SCP exception이 발생한 경우 이를 알려주는 플래그. MCR의 end of exception에 의하여 clear된다. 0 : No pending Exception 1 : Pending Exception Exist	0b
0	R	Reserved	0b

A.1.2 Master Command Register

SCPR15 (write only)

Bit	R/W	Description	Default Value
31 : 16	W	Reserved	-
15 : 11	W	Reserved	-
10 : 8	W	Reserved	-
7 : 6	W	Reserved	-
5 : 2	W	End of exception 0000 : instruction access violation 0001 : instruction TLB miss 0010 : privilege violation exception 0011 : data address misalignment 0100 : data access violation 0101 : data TLB miss 0110 : data write to read-only region 1000 : instruction address misalignment 1111 : N / A	-
1 : 0	W	Reserved	-

A.1.3 User Stack Pointer Register

SCPR13

Bit	R/W	Description	Default Value
31 : 2	RW	User Stack Pointer (USP)	0000_0000h
1 : 0	R	Always 0	00b

A.1.4 Vector Base Register

SCPR12

Bit	R/W	Description	Default Value
31 : 10	RW	Vector Base	00000h
9 : 0	R	Always 0	000h

A.1.5 Cache Control Register

SCPR11

Bit	R/W	Description	Default Value
31 : 7	W	Target Address[31:7]	-
6 : 4	W	Target Address[6:4] / Target Way	-
3	W	Operation 0 : address based invalidation 1 : way based invalidation	-
2	W	when LOCK = 0, write-back control 0 : without write-back 1 : with write-back if needed	-
	W	when LOCK = 1, Locking procedure 0 : Lock bit read 1 : cache locking	-
1	W	(LOCK) Lock Control 0 : Lock disable 1 : Lock enable	-
0	W	Cache Type 0 : Instruction Cache 1 : Data Cache	-

A.1.6 Memory Bank Register

SCPR9

Bit	R/W	Description	Default Value
31 : 28	RW	Memory Bank 7 Configuration 31 Memory Bank 7 TLB address translation 0 : TLB diable 1 : TLB enable 30 Memory Bank 7 access right (only valid at TLB disable) 0 : Supervisor only 1 : Supervisor / User 29 : 28 Memory Bank 7 Cache Configuration (only valid at TLB disable) 00 : Cache disable 01 : Reserved 10 : Cache enable with write-through 11 : Cache enable with write-back	0000b
27 : 24	RW	Memory Bank 6 Configuration	0000b
23 : 20	RW	Memory Bank 5 Configuration	0000b
19 : 16	RW	Memory Bank 4 Configuration	0000b
15 : 12	RW	Memory Bank 3 Configuration	0000b
11 : 8	RW	Memory Bank 2 Configuration	0000b
7 : 4	RW	Memory Bank 1 Configuration	0000b
3 : 0	RW	Memory Bank 0 Configuration	0000b

A.1.7 Sub-Bank Index/Configuration Register

SCPR8

Bit	R/W	Description	Default Value
31 : 7	R	Reserved	-
6 : 4	RW	Sub-bank Index 000 : sub-bank 0 001 : sub-bank 1 010 : sub-bank 2 011 : sub-bank 3 100 : sub-bank 4 101 : sub-bank 5 110 : sub-bank 6 111 : sub-bank 7	000b

Bit	R/W	Description	Default Value
3 : 0	RW	Sub-bank configuration (indexed by bit[6:4] 3 Configration information validity 0 : invalid 1 : valid 2 Access right 0 : supervisor only 1 : supervisor / user 1:0 cache configuration 00 : Cache disable 01 : Reserved 10 : Cache enable with write-through 11 : Cache enable with write-back	0000b

A.1.8 TLB Index Register

SCPR7

Bit	R/W	Description	Default Value
31 : 9	RW	Reserved	-
8	RW	TLB select separated TLB인 경우 어떤 TLB에 접근할 것인지 나타낸다. 0 : select instruction TLB / unified TLB 1 : select data TLB	0b
7 : 3	RW	TLB index 선택된 TLB에서 어떤 entry를 선택할 것인지 결정한다. TLB의 각 way는 최대 32entry를 지닐 수 있으며, 이 값은 TLB virtual address[16:12]도 사용된다.	00h
2 : 1	RW	TLB way 선택된 TLB에서 지정된 entry가 어떤 way에 존재하는지 지정한다. 각 TLB마다 최대 4way를 가질 수 있다.	00b
0	RW	SCPR6 property SCPR6에 입력되는 주소의 형식을 지정한다. 0 : virtual address register 1 : physical address register	0b

A.1.9 TLB Virtual Address Register

SCPR6

Bit	R/W	Description	Default Value
31 : 17	RW	TLB virtual address[31:17] virtual page number의 상위 부분을 지닌다. 하위 부분은 SCPR7에서 지정된 TLB index 부분을 사용한다.	00000h

Bit	R/W	Description	Default Value
16	R	Reserved	0b
15 : 8	RW	Process ID 가상 메모리에서는 각 프로세스마다 동일 가상 주소를 지닐 수 있으므로, 이를 process ID를 이용하여 구분한다. TLB에서는 총 256개의 process를 구분할 수 있으며, OS에서는 사용하는 프로세스 번호가 이 이상으로 할당되는 경우에는 TLB miss handler에서 TLB entry를 place할 때 적절하게 할당해서 사용하여야 한다.	00h
7	RW	Global access 0 : access disable 1 : access enable	0b
6	RW	read only page 해당 page가 read only 모드인지 나타낸다. 해당 page에 대한 write 접근은 access violation을 발생시킨다. 0 : read only page 1 : read / write	0b
5	R	Page dirty 해당 페이지의 데이터가 변경된 경우 설정된다. 0 : page is clean 1 : page is dirty	0b
4 : 3	RW	cache configuration 00 : cache disable 01 : reserved 10 : cache enable with write-through 11 : cache enable with write-back	00b
2	RW	access right 0 : supervisor only 1 : supervisor / user	0b
1	RW	accessed 해당 페이지가 접근되었는지 나타낸다. TLB miss의 경우 replacement를 수행할 때 어떤 way를 replace시킬지 결정하는데 사용될 수 있다. 0 : page is not accessed 1 : page is accessed	0b
0	RW	TLB line valid 해당 라인이 valid한지 설정한다. 특정 TLB 라인에 대하여 invalidation을 수행하거나, 초기 설정에서 사용된다. 0 : invalid page 1 : valid page	0b

A.1.10 TLB Physical Address Register

SCPR6

Bit	R/W	Description	Default Value
31 : 12	RW	TLB physical address[31:12] physical page number를 설정하기 위하여 사용된다.	00000h
11 : 0	RW	Reserved	000h

A.1.11 Sub-Bank Address Register

SCPR5

Bit	R/W	Description	Default Value
31 : 12	RW	Sub-bank Base Address Bit 31-12 Base Address = xxxx_x000h	00000h
11 : 0	RW	Sub-bank Size 0000_0000_0000 : 4 KB 0000_0000_0001 : 8 KB 0000_0000_0011 : 16 KB 0000_0000_0111 : 32 KB 0000_0000_1111 : 64 KB 0000_0001_1111 : 128 KB 0000_0011_1111 : 256 KB 0000_0111_1111 : 512 KB 0000_1111_1111 : 1 MB 0001_1111_1111 : 2 MB 0011_1111_1111 : 4 MB 0111_1111_1111 : 8 MB 1111_1111_1111 : 16 MB	000h

A.1.12 General Access Point Data Register

SCPR4

Bit	R/W	Description	Default Value
31 : 0	RW	General Access Point Data SCPR3에 의해 Index된 Register의 값을 쓰거나 읽을 수 있다.	-

A.1.13 General Access Point Index Register

SCPR3

Bit	R/W	Description	Default Value
31 : 0	RW	General Access Point Index GAP를 통해서 접근 할 수 있는 레지스터를 선택하는 index로 사용된다. GAP로 접근할 수 있는 레지스터의 종류는 refsec:gap 절을 참고하면 된다.	-

A.2 General Access Point

Number	R/W	Description	Remark
0x0000	R	Backup IR	
0x0001	R	Backup ER	
0x0002	R	Backup PC	
0x0010	R	EAD	
0x0300	R	Instruction TLB miss address	
0x0301	R	Data TLB miss address	
0x0302	R	TLB Process ID	
0x0500	R	Instruction cache lock condition	
0x0501	R	Data cache lock condition	
0x0600	R	Instruction Bus Error Address	
0x0601	R	Data Bus Error Address	
0x0700	RW	Global SPM Control Register	
0x0701	RW	Local iSPM Bank0 Control Register	
0x0702	RW	Local iSPM Bank0 Start Address Register	
0x0703	RW	Local iSPM Bank0 End Address Register	
0x0704	RW	Local dSPM Bank0 Control Register	
0x0705	RW	Local dSPM Bank0 Start Address Register	
0x0706	RW	Local dSPM Bank0 End Address Register	
0x0711	RW	Local iSPM Bank1 Control Register	
0x0712	RW	Local iSPM Bank1 Start Address Register	
0x0713	RW	Local iSPM Bank1 End Address Register	
0x0714	RW	Local dSPM Bank1 Control Register	
0x0715	RW	Local dSPM Bank1 Start Address Register	
0x0716	RW	Local dSPM Bank1 End Address Register	
0x0721	RW	Local iSPM Bank2 Control Register	
0x0722	RW	Local iSPM Bank2 Start Address Register	
0x0723	RW	Local iSPM Bank2 End Address Register	
0x0724	RW	Local dSPM Bank2 Control Register	
0x0725	RW	Local dSPM Bank2 Start Address Register	
0x0726	RW	Local dSPM Bank2 End Address Register	
0x0731	RW	Local iSPM Bank3 Control Register	
0x0732	RW	Local iSPM Bank3 Start Address Register	
0x0733	RW	Local iSPM Bank3 End Address Register	
0x0734	RW	Local dSPM Bank3 Control Register	
0x0735	RW	Local dSPM Bank3 Start Address Register	
0x0736	RW	Local dSPM Bank3 End Address Register	

A.2.1 Backup Register

GAP: 0x000

Bit	R/W	Description	Default Value
31 : 16	R	Reserved	0000h
15 : 0	R	Backup Instruction Register UII, UDI exception이 발생한 경우의 IR을 확인할 수 있다. NOP 1 명령 이후의 IR을 확인할 수 있다.	0000h

GAP: 0x001

Bit	R/W	Description	Default Value
31 : 0	R	Backup Extension Register UII, UDI exception이 발생한 경우의 ER을 확인할 수 있다. NOP 1 명령 이후의 ER을 확인할 수 있다.	0000_0000h

GAP: 0x002

Bit	R/W	Description	Default Value
31 : 0	R	Backup Program Counter Register UII, UDI exception이 발생한 경우의 PC를 확인할 수 있다. NOP 1 명령 이후의 PC를 확인할 수 있다.	0000_0000h

A.2.2 TLB

GAP: 0x300

Bit	R/W	Description	Default Value
31 : 0	R	Instruction TLB Miss Address Instruction TLB에 Miss가 발생한 경우의 address를 확인할 수 있다.	0000_0000h

GAP: 0x301

Bit	R/W	Description	Default Value
31 : 0	R	Data TLB Miss Address Data TLB에 Miss가 발생한 경우의 address를 확인할 수 있다.	0000_0000h

GAP: 0x302

Bit	R/W	Description	Default Value
31 : 8	R	Reserved	-
7 : 0	R	TLB Process ID TLB의 process ID를 확인할 수 있다.	00h

A.2.3 Cache

GAP: 0x500

Bit	R/W	Description	Default Value
31 : 4	R	Reserved	-
3 : 0	R	Instruction Cache Lock Condition Instruction Cache의 Lock 상태를 확인할 수 있다.	0h

GAP: 0x501

Bit	R/W	Description	Default Value
31 : 4	R	Reserved	-
3 : 0	R	Data Cache Lock Condition Data Cache의 Lock 상태를 확인할 수 있다.	0h

A.2.4 Bus

GAP: 0x600

Bit	R/W	Description	Default Value
31 : 0	R	Instruction Bus Error Address Instruction Bus의 Error가 발생한 Address를 확인할 수 있다.	0000_0000h

GAP: 0x601

Bit	R/W	Description	Default Value
31 : 0	R	Data Bus Error Address Data Bus의 Error가 발생한 Address를 확인할 수 있다.	0000_0000h

Appendix B

찾아보기