Core-A 마이크로프로세서의 코프로세서로 동작하는 AES 암호모듈의 하드웨어 설계

하창수* · 최병윤*

Hardware Design of AES Cryptography Module Operating as Coprocessor of Core-A Microprocessor

Chang-Soo Ha* · Byeong-Yoon Choi*

요 약

Core-A 마이크로프로세서는 32-bit RISC 구조의 국산 임베디드 마이크로프로세서로서 특허청의 지원을 받아 KAIST의 주관아래 개발된 프로세서이다. 본 논문에서는 Core-A 마이크로프로세서와 코프로세서간의 인터페이스 방안에 대하여 분석하고 효율적인 구조를 제안 한다. 인터페이스 방안의 검증을 위해 코프로세서로 사용된 AES 암호 프로세서는 128-bit의 키와 블록을 갖는 대칭키 암호 알고리즘이다. 코프로세서 인터페이스 회로와 AES 암호 프로세서는 Verilog-HDL로 작성되었으며, Modelsim 시뮬레이터를 사용하여 시뮬레이션을 수행하였다. 삼성 0.35um CMOS 표준 셀 라이브러리를 사용하여 AES를 제외한 코프로세서 인터페이스 부분을 합성한 결과 약 90Mhz의 동작 주파수를 가지며, 3743개의 게이트 수로 구성되었다. 본 논문에서 구현한 코프로세서 인터페이스 회로는 Core-A와 코프로세서간의 효율적인 명령어 및 데이터 전달을 수행할 수 있다.

ABSTRACT

Core-A microprocessor is the all-Korean product designed as 32-bit embedded RISC microprocessor developed by KAIST and supported by the Industrial Property Office. This paper analyze Core-A microprocessor architecture and proposes efficient method to interface Core-A microprocessor with coprocessor. To verify proposed interfacing method, the AES cryptography processor that has 128-bit key and block size is used as a coprocessor. Coprocessor and AES are written in Verilog-HDL and verified using Modelsim simulator. It except AES module consists of about 3,743 gates and its maximum operating frequency is about 90Mhz under 0.35um CMOS technology. The proposed coprocessor interface architecture is efficiency to send data or to receive data from Core-A to coprocessor.

키워드

Core-A, 코어-에이, 코프로세서, AES

Key word

Core-A, Coprocessor, AES

* 동의대학교 컴퓨터공학과

접수일자 : 2009. 06. 27
심사완료일자 : 2009. 07. 20

Ⅰ. 서 론

오늘날 휴대전화, MP3 플레이어, 차량용 네비게이션 시스템등 임베디드 시스템 시장이 빠른 속도로 성장하여 임베디드 시스템에 사용되는 마이크로프로세서의 중요성이 증가하게 되었다. 현재 외산 임베디드 프로세서들이 시장을 선점하여 다양한 분야에 적용되고 있으며, 앞으로도 임베디드 시스템은 더욱 필요로 해 질 것이므로 국산 임베디드 마이크로프로세서의 개발이 필요한 상황이다.

특허청에서는 이러한 요구사항에 따라 32-bit RISC 구조의 Core-A 임베디드 마이크로프로세서를 개발하여 발표하였다[1]. 또한, Core-A 마이크로프로세서를 기반으로 한 응용프로그램 작성을 위한 어셈블러와 컴파일러 및 디버거를 함께 개발하여 배포하고 있으며, eclipse를 기반으로 한 IDE(Integrated Development Environment) 환경도 지원하고 있다[2]. Core-A는 합성 가능한 soft-IP로 설계되어 있기 때문에 응용하려는 임베디드 시스템 환경에 맞게 최적화 시키거나 필요한 기능을 추가할 수 있다[1]. 기존에 설계된 IP들을 결합하기 위한 시스템 버스로서 Core-B 버스와 AMBA 버스, Wishbone 버스를 지원하며 캐쉬와 MMU를 적용할 수 있도록 하여 Core-A를 활용한 임베디드 시스템 개발이 용이 하도록 하였다[3],[4],[5],[6],[7]. 오늘날의 임베디드 시스템 환경에서는 대량의 멀티미디어 데이터를 처리하기 위한 DSP 또는 유비쿼터스 환경을 지원하기위한 통신 프로세서, 정보보호를 위한 암호 프로세서 등이 함께 사용되기 때문에 이를 지원하기 위한 코프로세서 환경이 필요하며, Core-A에서는 최대 4개의 코프로세서를 지원할 수 있다[1]. 본 논문에서는 Core-A의 코프로세서 인터페이스 방안을 연구하여 Verilog-HDL로 구현 하였으며 AES 암호 프로세서를 코프로세서로 적용하여 올바른 동작을 검증 하였다[8].

본 논문의 구성은, 2장에서 기본적인 Core-A의 구조와 특징을 소개하고, 3장 코프로세서를 위한 Core-A의 버스 인터페이스 구조에서는 Core-A 프로세서가 가지고 있는 코프로세서 인터페이스를 분석한다. 4장의 코프로세서 인터페이스 설계에서는 3장의 분석 내용을 기반으로 Core-A가 코프로세서와 인터페이스 할 수 있는 구조를 제안한다. 5장의 AES 코프로세서 설계에서는 Core-A의 코프로세서로 동작하기 위한 인터페이스를 가진 AES 암호 프로세서에 대한 내용을 기술한다. 6장의 코프로세서 인터페이스와 AES의 통합 설계 검증 및 시뮬레이션에서는 Core-A의 어셈블러와 Modelsim 시뮬레이터를 사용하여 설계한 코프로세서 인터페이스 구조가 올바른 동작을 수행하는지 검증하였다. 마지막으로 7장에서는 결론을 맺는다.

Ⅱ. Core-A 구조

Core-A는 32-bit 임베디드 RISC 프로세서로서 데이터 메모리와 명령어메모리가 분리된 하버드 구조이다. 각 메모리마다 캐쉬와 MMU를 적용할 수 있으며 로컬 버스를 통해 코프로세서와 ASR(Application Specific Registers)에 접근할 수 있다. 주변 장치와의 데이터 전송을 위한 시스템 버스로서 Core-B가 사용되며 간단한 글루 로직을 추가하여 AHB 버스와 연결 가능하다. Core-A의 전체적인 구조를 그림 1에 나타내었다[1].

Core-A는 5단계의 파이프라인 구조를 가지며 각 파이프라인 단계는 명령어 인출, 명령어 디코딩, 실행, 메모리 연산, 레지스터 쓰기로 이루어진다. Core-A는 16개의 GPR(General Purpose Registers)과 프로그램 카운터, 프로그램 상태 레지스터를 가지고 있으며, 216개의 ASR 영역을 사용할 수 있다. GPR의 마지막 3개의 레지스터는 예외가 발생했을 때와 함수 호출의 복귀 주소를 담기 위한 용도로 사용된다. ASR은 Core-A의 로컬 버스로 접근 할 수 있기 때문에 응용프로그램이 외부 메모리를 접근 하는 것보다 훨씬 빠른 속도로 데이터를 읽고 쓸 수 있으며, 외부 메모리의 용량이 제한되는 임베디드 시스템의 단점을 보완할 수 있는 역할을 한다.

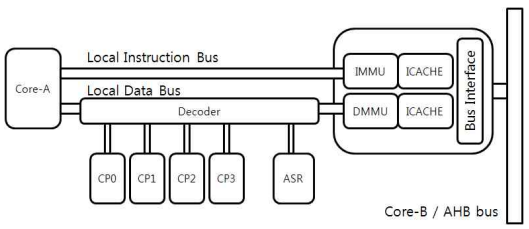


그림 1. Core-A 구조
Fig. 1. Core-A Architecture

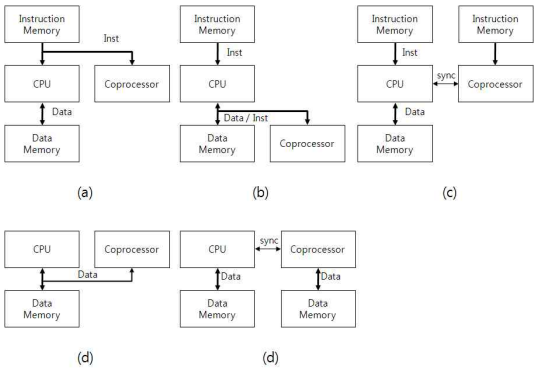


그림 2. 코프로세서의 구성 방법
Fig. 2. Configuration Methods for Coprocessor

또한, 본 연구의 주제인 코프로세서와 관련된 부분으로서 각각의 코프로세서가 사용할 수 있는 레지스터도 16개가 개별적으로 준비되어 있으며 하나의 레지스터 크기는 32-bit이다.

Core-A 프로세서는 기존의 임베디드 프로세서들의 장점들을 적용하였다. 그 중 프로그래밍 가능한 지연 슬롯과 분기 슬롯은 명령어 코드의 양을 크게 줄일 수 있다. 또한 조건적인 할당과 분기 연산자를 지원하며 쉬프트 연산과 함께 덧셈이 수행 되는 구조를 갖는다. 마지막으로 완전히 합성가능한 soft-IP라는 것도 Core-A의 특징이라고 할 수 있다.

Ⅲ. 코프로세서를 위한 Core-A의 버스 인터페이스 구조

코프로세서는 CPU의 기능을 보완하는 역할을 하며 CPU와 효율적으로 정보를 주고받을 수 있어야 한다. 그림 2에 CPU와 코프로세서가 구성되는 일반적인 방법들을 나타내었다.

Core-A는 명령어 메모리로부터 전달된 명령어를 Core-A가 해석하여 코프로세서가 수행해야할 명령이라고 판단되면 로컬 버스를 통해 코프로세서로 전달하는 (b)의 구조이다. 따라서 Core-A는 로컬 버스를 통해 코프로세서가 수행해야할 명령을 전달하고 명령어 수행에 필요한 데이터 역시 로컬 버스를 통해 전송하게 된다. 그림 3에 Core-A의 핀 정보를 나타내었다.

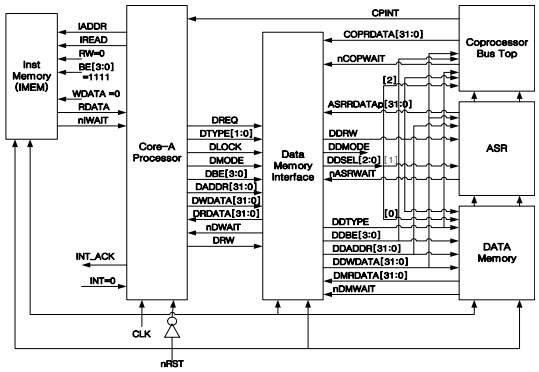


그림 3. Core-A 핀 정보
Fig. 3. Core-A Pin Description

Core-A 프로세서는 자신의 로컬 버스를 통해 코프로세서와 연결되며 데이터 메모리와 ASR이 코프로세서와 하나의 로컬 버스를 공유한다. 표1에 코프로세서와 관련된 핀과 기능을 나타내었다.

표 1. 코프로세서와 관련된 핀과 기능
Table 1. Function of Pins about Coprocessor

이름	소스	기능
DREQ	Core	Data bus request
DADDR	Core	Data bus byte address
DRW	Core	Data bus write enable
DLOCK	Core	Data bus lock
DTYPE	Core	Data bus request type 00 : memory 01 : ASR 10 : coprocessor data 11 : coprocessor instruction
DMODE	Core	Processor mode 0 : user mode 1 : kernel mode
DBE	Core	Data bus write byte enable
DWDATA	Core	Data to data bus
DRDATA	D.M.	Data from data bus
DFAULT	D.M.	Data memory fault
nDWAIT	D.M.	Data bus ready
CPINT	C.P.	Coprocessor exception

데이터 버스와 코프로세서 버스가 공유되기 때문에 표1에 표기된 데이터 버스는 코프로세서 버스로 해석한다. DTYPE 신호를 통해 공유된 버스의 대상을 결정하며

DWDATA와 DRDATA를 통해 코프로세서의 명령어와 데이터를 주고받는다. Core-A와 코프로세서간의 동작을 제어하기 위해 nDWAIT와 CPINT를 사용할 수 있다.

코프로세서와 통신을 하기 위해 Core-A로부터 나오는 데이터는 로컬 버스 디코더(그림 3의 Data Memory Interface 모듈)를 통과하게 된다.

디코더를 통과한 데이터는 DTYPE의 값에 따라 코프로세서의 명령어 또는 코프로세서의 데이터로 해석되어 코프로세서로 전달된다. DTYPE 값을 디코딩하여 만든 CPSEL 신호는 4개의 코프로세서 중 하나를 선택하는 역할을 하며 CPTYPE 신호는 현재 전송되는 데이터가 코프로세서의 명령어인지 데이터인지를 결정한다. 코프로세서로부터 나오는 값은 CPRDATA를 통해 전달되며 nCPWAIT와 CPINT 신호를 Core-A로 보낼 수 있다. 4개의 코프로세서 중 하나를 선택하여 Data Memory Interface와 데이터를 주고받을 수 있도록 해 주는 역할을 수행하기 위해 버스의 형태로 구성된다. 일반적으로 버스의 구현은 3-상태 버퍼를 이용한 구성 또는 멀티플렉서를 활용하여 구성할 수 있는데 Core-A에서는 멀티플렉서를 활용하여 구현하였다.

Ⅳ. 코프로세서 인터페이스 설계

코프로세서 버스를 통과한 값들은 COP_CORE_0~3 모듈에 연결되어 코프로세서로 전송된다. 그림 4에 COP_CORE의 구조를 나타내었다. 그림 4의 COP_CORE는 CP_CTRL모듈과 CP_TOP 모듈로 구성된다. CP_TOP 모듈은 실제 사용될 코프로세서가 들어가는 부분이며 CP_CTRL 모듈은 코프로세서를 직접 제어하게 된다.

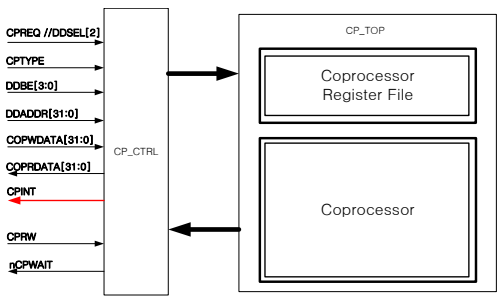


그림 4. COP_CORE의 모듈
Fig. 4. COP_CORE Module

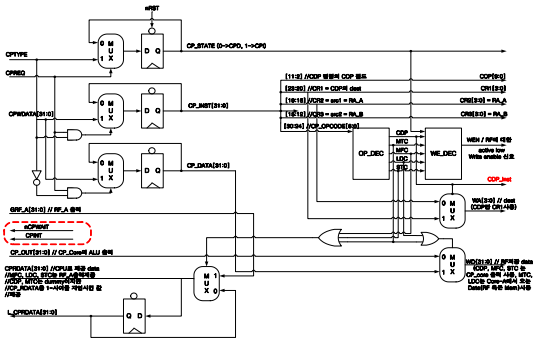


그림 5. 제안한 CP_CTRL의 모듈
Fig. 5. The Proposed CP_CTRL Module

CP_CTRL 모듈은 코프로세서에 기본적으로 필요한 명령어를 전달하고 코프로세서의 메모리 접근에 필요한 읽기 주소와 쓰기 주소 및 쓰기 데이터를 전달해야한다. 또한 코프로세서로부터 나오는 읽기 데이터와 nCPWAIT 신호, CPINT신호를 Core-A로 전달해야 한다. 그림 5에 본 논문에서 제안하는 CP_CTRL의 구조를 나타내었다. 그림 5의 CP_CTRL 모듈에서 CPWDATA는 CPTYPE 값에 따라 코프로세서의 명령어(CP_INST) 또는 쓰기 데이터(CP_DATA)로 제공되며 CPTYPE이 1인 경우 CP_INST는 CDP(Coprocessor Data Processing) 명령어 형식으로 해석되어 코프로세서로 전달된다. 그림 6에 CDP 명령의 형식을 나타내었다. CDP 명령은 코프로세서에게 필요한 명령을 전달하기 위해 사용되는 어셈블리 명령이다. CDP[31]은 프로그래밍 가능한 지연 슬롯을 사용할 수 있게 해 주는 것으로 NOP(no operation)연산의 삽입 유무를 나타내는 필드이다. CDP[30:24]는 CDP 명령임을 나타내기 위한 op-code 부분이며 이것은 Core-A가 명령을 디코딩 할 때 사용된다. CDP[23:20], CDP[19:16], CDP[15:12]는 각각 코프로세서 연산의 목적지 레지스터, 첫 번째 소스레지스터, 두 번째 소스레지스터를 나타내며, CDP[11:2]는 COP 필드로써 코프로세서의 op-code가 된다. CDP[1:0]은 명령이 수행 될 코프로세서의 번호를 나타낸다. CP_CTRL 모듈에서 생성된 개별적인 필드 값들은 코프로세서가 전달 받아 적절한 연산을 수행하게 된다.

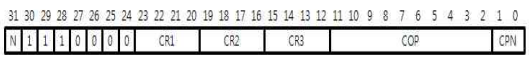


그림 6. CDP 명령어 형식
Fig. 6. CDP Instruction Format

2장에서 설명한 것처럼 코프로세서가 사용할 수 있는 레지스터 공간은 2^4이므로 총 32-bit * 16의 크기를 사용할 수 있다. 사용되는 코프로세서의 특성에 따라 제공된 레지스터 공간이 부족할 수도 있기 때문에 레지스터를 효율적으로 운영해야 하며 본 연구에서는 16개의 레지스터 공간을 두 개로 분할하여 각각 8개의 읽기와 쓰기 전용 레지스터로 구성하고 코프로세서의 동작을 제어하는 모듈을 추가로 구성하여 응용환경에 필요한 만큼의 데이터를 제공할 수 있도록 하였다. 그림 7에 본 논문에서 제안하는 CP_TOP 모듈의 블록도를 나타내었다. 그림 7의 CP_CON 모듈은 코프로세서에게 필요한 명령을 제공하며, 만약 코프로세서가 쓰기 레지스터의 크기보다 더 많은 양의 데이터를 입력으로 필요로 할 경우 쓰기 레지스터 크기만큼의 데이터를 여러 번 가져 갈 수 있도록 제어한다. 또한 코프로세서에서 출력되는 결과의 크기가 읽기 레지스터보다 더 많을 경우에도 읽기 레지스터를 여러 번 접근하여 필요한 만큼의 데이터를 전송할 수 있다. 따라서 이렇게 구성된 레지스터는 다양한 종류의 코프로세서를 지원할 수 있으며 제한된 레지스터의 크기에 영향을 받지 않고 원하는 크기의 데이터를 주고받을 수 있다.

앞에서 설명한 코프로세서 인터페이스 환경을 기반으로 Core-A가 코프로세서와 상호 연동하여 연산을 수행할 수 있다. Core-A가 nDWAIT와 CPINT 신호를 지원하기 때문에 코프로세서와 상호 연동할 수 있는 방법은 크게 세 가지가 있다. 첫째, Programmed-I/O 방식. 둘째, nDWAIT를 활용한 Pipeline-stall 방식. 셋째, CPINT를 활용한 인터럽트 방식이다.

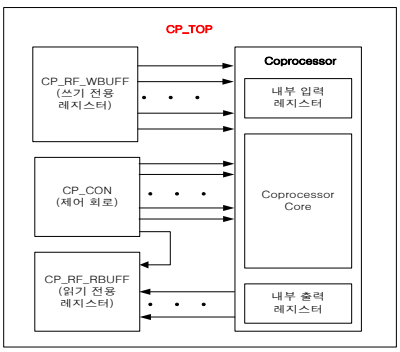


그림 7. 제안한 CP_TOP 모듈
Fig. 7. The Proposed CP_TOP Module

각 방식에 따라 CP_CTRL 회로와 CP_TOP회로의 변경이 필요하며 소프트웨어 코딩 방식 또한 달라져야 한다. 본 논문에서는 CPINT 신호를 활용한 인터럽트 방식의 코프로세서 인터페이스 방안에 대해서만 기술한다.

Ⅴ. AES 코프로세서 설계

Core-A의 코프로세서 인터페이스 방안을 검증하기 위해 사용한 코프로세서는 AES 암호 프로세서이다. AES는 Advanced Encryption Standard의 약자로 미국 정부 표준으로 지정된 블록 암호 알고리즘이다[8]. 기존의 DES(Data Encryption Standard) 암호 알고리즘이 CPU 성능의 향상으로 안전성이 떨어지게 되자 DES를 대체할 목적으로 미국 표준 기술 연구소(NIST)가 5년의 표준화 과정을 거쳐 2001년 11월 26일에 연방 정보 처리 표준(FIPS 197)으로 발표한 알고리즘이다. 2002년 5월 26일부터 효력을 발휘하기 시작하였으며 현재까지 다양한 분야에서 활용되고 있다. AES는 벨기에의 암호학자인 존 대먼과 빈센트 라이먼에 의해 만들어졌으며 두 사람의 이름을 합해 레인달(Rijndael)로 명명되었다. 레인달 알고리즘은 키와 블록의 크기로 128, 160, 192, 256-bit를 사용할 수 있지만 AES에서는 키는 128, 192, 256-bit, 블록은 128-bit만을 표준으로 인정한다. 본 논문에서는 구현의 단순화를 위해 키의 길이는 128-bit만을 지원하도록 하였다(AES-128). 그림 8에 ASM(Algorithmic State Machine) 도표의 일부를 나타내었다.

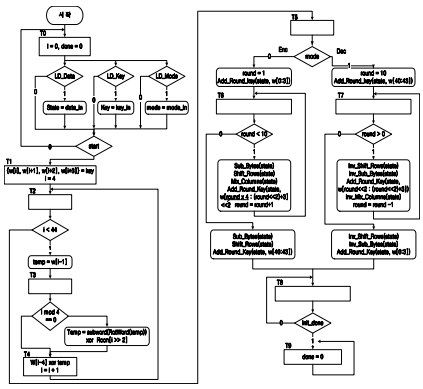


그림 8. AES의 ASM 도표
Fig. 8. AES ASM Chart

그림 8에 나타낸 ASM 도표는 AES 알고리즘의 구현 외에 코프로세서 인터페이스를 위한 상태들을 함께 정의하였다. 우선 T0 상태에 나타낸바와 같이 LD_Key, LD_Data, LD_Mode 제어 신호에 따라 코프로세서 레지스터 파일로부터 AES 내부 버퍼로 데이터를 전송하는 상태가 필요하다. 이것은 16개로 제한되는 코프로세서 레지스터를 무한개의 레지스터처럼 사용할 수 있도록 해 준다. T8 상태의 init_done 신호는 인터럽트 핸들러가 제공해야 하는 제어 신호로서 인터럽트를 발생시키는 done 신호를 비활성화 시킨다. AES의 done 신호에 의해 인터럽트를 받은 Core-A가 인터럽트 제공 신호를 비활성화 시키지 않으면 인터럽트를 처리한 후에도 같은 인터럽트가 계속 뜨기 때문에 이러한 처리가 필요하다. 코프로세서 인터페이스를 위한 제어신호들을 반영하여 설계한 AES 프로세서의 블록도를 그림 9에 나타내었다.

설계한 AES 알고리즘은 한 라운드에 한 클럭을 사용하는 구조이며, 각 라운드마다 사용되는 서브키는 on-the-fly 방식으로 제공된다. 단, 복호화시에는 주어진 복호키로부터 서브키를 직접 생성할 수 없기 때문에 키 스케줄의 사전 연산이 필요하다. 설계한 AES-128을 Modelsim으로 검증한 결과를 그림 10와 그림 11에 나타내었다.

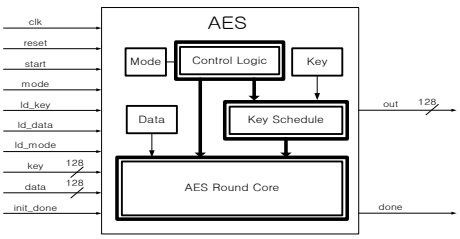


그림 9. AES 블록도
Fig. 9. AES Block Diagram

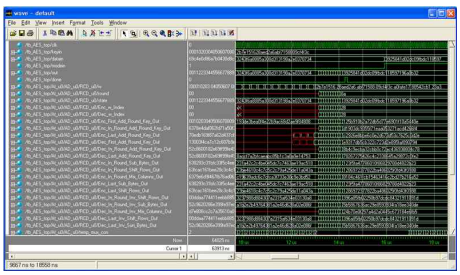


그림 10. AES-128 암호화의 Modelsim 시뮬레이션
Fig. 10. Modelsim Simulation of AES-128 Encryption

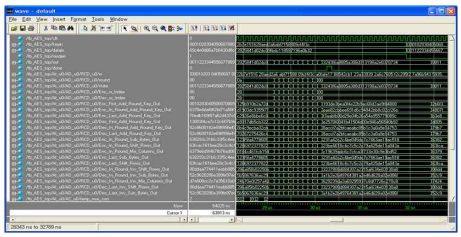


그림 11. AES-128 복호화의 Modelsim 시뮬레이션
Fig. 11. Modelsim Simulation of AES-128 Decryption

VI. 코프로세서 인터페이스와 AES의 통합 설계 검증 및 시뮬레이션

3장에서 제안한 코프로세서 인터페이스 방안 과 4장에서 설계한 AES 알고리즘의 통합 설계를 위해 그림 12에 나타낸 AES 코프로세서 인터페이스 구조를 제안한다. 그림 12의 CP_AES_TOP 모듈은 쓰기 레지스터인 CP_RF_WBUFF_AES, 읽기 레지스터인 CP_RF_RBUFF_AES과 CP_AES_CON 및 AES 모듈로 구성된다. AES 코프로세서 인터페이스 회로에서 사용할 레지스터 파일과 AES 내부 버퍼의 사상은 표 2와 같다.

쓰기 레지스터는 AES 프로세서에서 사용될 키와 데이터 공간을 공유하고 있으며 CP_AES_CON 모듈에서 생성되는 제어신호에 따라 AES 내부의 키와 데이터 버퍼로 블록단위 복사가 수행된다. 읽기 레지스터에는 AES의 연산이 완료 되고 난 후의 결과를 저장하기 위한 제어 신호 입력이 필요하며 CP_AES_CON 모듈에서 생성된다.

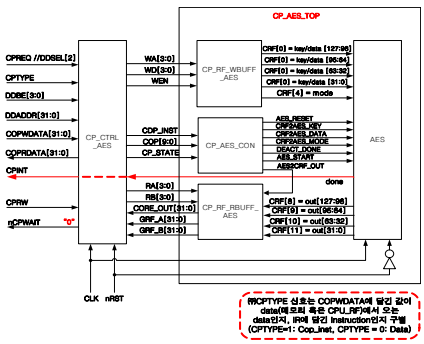


그림 12. 제안한 AES 코프로세서 인터페이스 구조
Fig. 12. The Proposed Architecture of AES Coprocessor Interface

표 2. AES 내부 버퍼와 레지스터 사상
Table 2. Mapping Table of AES Buffer and Register File

CP_RF[#]	AES 내부 버퍼	특성
CP_RF[0]	key/data [128:96]	쓰기전용
CP_RF[1]	key/data [95:64]	쓰기전용
CP_RF[2]	key/data [63:32]	쓰기전용
CP_RF[3]	key/data [31:0]	쓰기전용
CP_RF[4]	mode	쓰기전용
.....	Reserved	
CP_RF[8]	out [128:96]	읽기전용
CP_RF[9]	out [95:64]	읽기전용
CP_RF[10]	out [63:32]	읽기전용
CP_RF[11]	out [31:0]	읽기전용
.....		
CP_RF[15]	done // Dummy	읽기전용

AES 모듈에서는 연산의 완료를 알리는 done 신호가 나오게 되는데, 이것은 Core-A로 가는 인터럽트 신호로 사용하게 된다. Core-A가 인터럽트를 감지하여 인터럽트 핸들러를 수행하게 되는데 이때, 인터럽트 핸들러에서는 init_done 신호를 발생하는 명령을 전달하여 done 신호를 비활성화 시킨다. 그림 13에 CP_RF_WBUFF_AES의 구조를 나타내었다. 본 논문의 AES-128에서 필요한 쓰기 레지스터의 크기는 5개이며 각 레지스터의 값이 병렬로 출력되는 구조로 설계하였다. mode는 1-bit만 사용되기 때문에 CRF[4]의 최하위비트만을 담아간다.

그림 14에는 CP_RF_RBUFF_AES의 구조를 나타내었다.

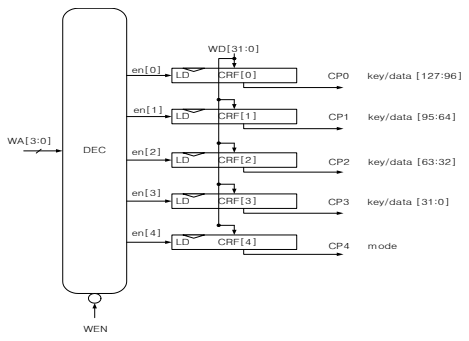


그림 13. CP_RF_WBUFF_AES 모듈
Fig. 13. CP_RF_WBUFF_AES Module

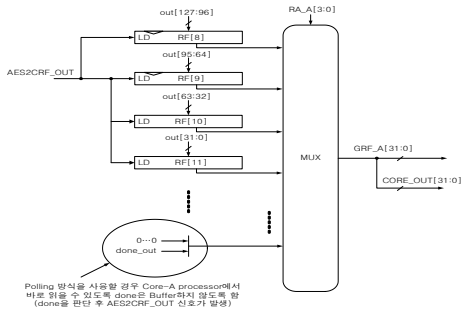


그림 14. CP_RF_RBUFF_AES 모듈
Fig. 14. CP_RF_RBUFF_AES Module

본 논문의 AES-128에서 필요한 읽기 레지스터는 4개이며 각 레지스터의 입력은 AES2CRF_OUT 제어신호의 입력에 따라 AES-128의 출력이 동시에 담기도록 하였다. 폴링 방식을 사용할 경우 done 신호를 읽어가기 위해 읽기 레지스터를 사용하게 되는데 이때, Core-A가 바로 읽어 갈 수 있도록 버퍼에 담지 않거나 AES2CRF_OUT 제어 신호에 상관없이 매 클럭마다 done의 값을 담도록 해야 한다. 그림 15에는 CP_AES_CON 모듈을 나타내었다. CP_AES_CON 모듈은 CDP 명령의 COP 필드를 해석하여 자신을 제외하 CP_AES_TOP 모듈의 나머지 모듈들에게 제어신호를 제공한다. 현재 CDP 명령이 유효한지를 결정하기 위해 CP_STATE 신호를 함께 검사해야 하며 한 클럭사이클 동안만 제어신호 생성을 보장하기 위해 SPG(Single Pulse Generation) 모듈을 통과하도록 하였다. CP_AES_CON 모듈은 7개의 내부 제어 신호를 생성하며 이 중 6개는 AES 모듈로 전달되며 AES2COP_OUT 신호만 읽기 레지스터로 전달된다.

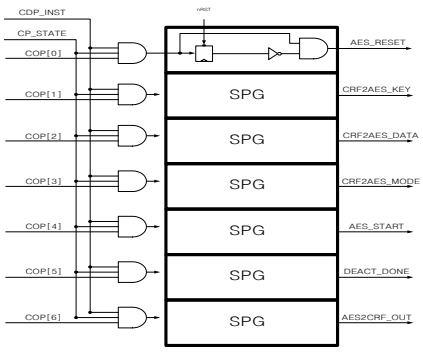


그림 15. CP_AES_CON 모듈
Fig. 15. CP_AES_CON Module

표 3에 AES 동작을 위한 CDP 명령의 인코딩 내용을 나타내었다. 각 명령의 동작은 다음과 같다.

- AES_RESET
  - AES local reset
- CRF2AES_KEY
  - key={CRF[0], CRF[1], CRF[2], CRF[3]}
- CRF2AES_DATA
  - data={CRF[0], CRF[1], CRF[2], CRF[3]}
- CRF2AES_MODE
  - mode=CRF[4]
- AES_START
  - AES Operation start
- DEACT_INT
  - Deactivate done signal
- AES2CRF_OUT
  - {CRF[8], CRF[9], CRF[10], CRF[11]}=out
  - CRF[15]=done(Programmed I/O)

표 3에서 볼 수 있듯이 CP_AES_CON 모듈에서 디코딩 회로의 단순화를 위해 one-hot 인코딩 기법을 사용하였다. 제안한 코프로세서 인터페이스를 테스트하기 위한 방안은 데이터메모리 사용 유무에 따라 크게 두 가지 방안으로 구분할 수 있다. 첫째 방안은 데이터메모리를 사용하지 않고 코프로세서로 전달할 데이터를 즉치 값으로 생성하여 전달하는 것이다. 이 방법은 데이터메모리에 접근하지 않아도 되기 때문에 LDC, STC 명령을 사용하지 않는다.

표 3. AES 동작을 위한 CDP 명령의 인코딩
Table 3. Encoding of CDP Instruction for AES Operation

COP[9:0]	명령
00_0000_0001	AES_RESET
00_0000_0010	CRF2AES_KEY
00_0000_0100	CRF2AES_DATA
00_0000_1000	CRF2AES_MODE
00_0001_0000	AES_START
00_0010_0000	DEACT_INT
00_0100_0000	AES2CRF_OUT
others	Reserved

표 4. 데이터메모리의 내용
Table 4. Contents of Data memory

주소(16진수)	데이터(16진수)
0	00010203
4	04050607
8	08090a0b
C	0c0d0e0f
10	00112233
14	44556677
18	8899aabb
1C	ccddeeff
20	00000000

둘째 방안은 데이터메모리에 코프로세서로 전달할 데이터를 적재한 후 메모리 접근 명령을 사용하는 것이다. 데이터메모리를 사용하는 방법이 현실적이기 때문에 본 연구에서는 데이터메모리를 사용하여 인터페이스 구조를 검증하였다. 표 4는 데이터메모리에 들어가는 데이터를 주소별로 나타낸다. 0~C 번지의 값은 AES의 key이며 10~1C번지의 값은 data이다. 20번지의 값은 암호화/복호화를 나타내는 mode 값이며 0은 암호화 동작을 나타낸다.

명령어 메모리에 들어갈 코드는 어셈블리어로 작성하며 그림 16에 암호화 연산을 수행하기 위한 어셈블리어 코드를 나타내었다.

```
@ Encryption
    MOVI   r0, 0x0

    LDC CP0, c0, r0(0x0) @ 0x00010203 key[127:96]
    LDC CP0, c1, r0(0x1) @ 0x04050607 key[95:64]
    LDC CP0, c2, r0(0x2) @ 0x08090a0b key[63:32]
    LDC CP0, c3, r0(0x3) @ 0x0c0d0e0f key[31:0]
    CDP CP0.CRF2AES_KEY c7, c0, c1 @ CRF2AES_KEY

    LDC CP0, c0, r0(0x4) @ 0x00112233 data[127:96]
    LDC CP0, c1, r0(0x5) @ 0x44556677 data[95:64]
    LDC CP0, c2, r0(0x6) @ 0x8899aabb data[63:32]
    LDC CP0, c3, r0(0x7) @ 0xccddeeff data[31:0]
    CDP CP0.CRF2AES_DATA c7, c0, c1 @ CRF2AES_DATA

    LDC CP0, c4, r0(0x8) @ 0x00000000 mode = Encryption
    CDP CP0.CRF2AES_MODE c7, c0, c1 @ CRF2AES_MODE

    CDP CP0.AES_RESET c7, c0, c1 @ AES_RESET
    CDP CP0.AES_START c7, c0, c1 @ AES_START
    BR r15
```

그림 16. 암호화를 수행하기 위한 어셈블리어 코드
Fig. 16. Assembly Language Codes for AES Encryption

```
INT_HANDLER:
    CDP CP0.DEACT_INT c7, c0, c1 @ DEACT_INT
    CDP CP0.AES2CRF_OUT c7, c0, c1 @ AES2CRF_OUT

    STC     CP0, c8, r0(0x9)
    STC     CP0, c9, r0(0xa)
    STC     CP0, c10, r0(0xb)
    STC     CP0, c11, r0(0xc)
    RFI
```

그림 17. 인터럽트 핸들러를 위한 어셈블리어 코드
Fig. 17. Assembly Language Codes for Interrupt Handler

또한 인터럽트 핸들러에서 수행할 어셈블리어 코드를 그림 17에 나타내었다. 그림 16와 17에 나타낸 어셈블리어 코드는 Core-A의 어셈블러를 사용하여 기계어로 번역이 가능하다. 단, Core-A의 어셈블러는 코프로세서에서 제공하는 명령어가 무엇이 있는지 알지 못하기 때문에 CDP 명령의 COP 필드는 어셈블러로 번역 후 명령을 직접 수정해야 하는 과정이 필요하다. 이것은 표준화된 코프로세서 명령 인터페이스가 정의되어 있지 않기 때문이며 어셈블러의 사용에 제약이 되는 부분이기도 하다. 본 연구에서는 명령어메모리에 들어갈 전체코드를 어셈블러의 도움을 받아 생성한 후 CP_AES_CON 모듈의 인코딩 환경에 따라 CDP 명령어의 COP 필드를 직접 수정하여 테스트를 수행하였다. Modelsim 시뮬레이터를 사용하여 제안한 Core-A 코프로세서 인터페이스 환경을 검증한 결과를 그림 18에 나타내었다.

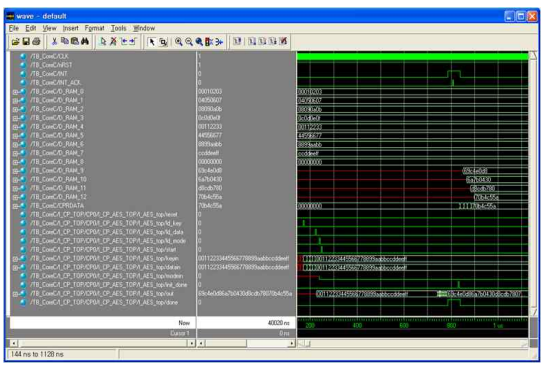


그림 18. Core-A 코프로세서 인터페이스 환경의 Modelsim 시뮬레이션 결과
Fig. 18. Modelsim Simulation Result of Core-A Coprocessor Interface Circuits

그림 18의 시뮬레이션 결과를 살펴보면, done 신호가 활성화 되면 Core-A의 인터럽트 신호가 활성화 되어 인터럽트 핸들러에 의해 AES의 연산결과가 데이터메모리로 옮겨지는 것을 확인할 수 있다.

표 5. 코프로세서 인터페이스 방식에 따른 암호화 명령어 개수
Table 5. The Number of Instructions for Four Coprocessor Interface Methods.

데이터메모리 사용여부	구현 방안	TEA	DES	AES
사용 않함	Programmed I/O	30	28	44
	Pipeline stall	27	25	41
	Interrupt	28	26	42
사용 함	Interrupt	14	13	21

제안한 코프로세서 인터페이스 방안을 사용하여 각각의 방식으로 코프로세서를 운용할 때 필요한 코드의 크기를 표 5에 나타내었다. 표 5는 코프로세서 인터페이스 방식에 따라 TEA, DES, AES 암호 프로세서를 적용하였을 경우 사용되는 명령어의 수를 나타내고 있다. 데이터메모리를 사용하는 방법이 데이터메모리를 사용하지 않는 방법보다 약 50%의 명령어코드를 절약할 수 있다. 하지만 데이터메모리를 사용하는 경우에는 메모리 접근 연산이 많이 사용되기 때문에 구현된 시스템의 메모리 성능에 따라 코프로세서와 관련된 연산 속도가 달라질 수 있다. 데이터메모리 사용 유무와 관계없이 인터럽트를 사용한 방법이 가장 적은 수의 명령어 코드가 필요함을 알 수 있으며 본 논문에서 제안한 코프로세서 인터페이스 구조가 다양한 방법으로 구현 가능한 것을 확인하였다. 그리고, AES를 제외한 나머지 부분을 삼성 0.35um CMOS 표준 셀 라이브러리를 사용하여 합성한 결과 제안한 코프로세서 인터페이스 구조는 약 90Mhz의 동작 속도를 가지며, 3743개의 게이트 수로 구성된다.

Ⅶ. 결 론

본 논문에서는 국산 32-bit 임베디드 RISC 마이크로프로세서인 Core-A의 코프로세서 인터페이스 구조를

제안하고 AES-128 프로세서를 코프로세서로 사용하여 올바른 동작을 검증하였다. 제안한 코프로세서 인터페이스 구조는 약간의 수정으로 Programmed I/O 방식, Pipeline stall 방식, Interrupt 방식 모두를 지원할 수 있으며 코프로세서가 요구하는 레지스터의 크기에 상관없이 효율적인 데이터 전달이 가능하다.

인터페이스 회로와 AES 암호 프로세서는 Verilog-HDL로 작성되었으며, Modelsim 시뮬레이터를 사용하여 시뮬레이션을 수행하였다. AES를 제외한 나머지 부분을 삼성 0.35um CMOS 표준 셀 라이브러리를 사용하여 합성한 결과 약 90Mhz의 동작 속도를 가지며, 3743개의 게이트 수로 구성된다. 따라서, 본 논문에서 구현한 코프로세서 인터페이스 회로는 Core-A와 코프로세서간의 효율적인 명령어 및 데이터 전달을 수행하므로 코프로세서가 필요한 신호처리 또는 멀티미디어 데이터처리를 수행하는 임베디드 시스템 환경에 적용될 수 있다고 판단된다.

향후 연구과제로는 본 논문에서 다룬 코프로세서 인터페이스 방안보다 범용으로 사용될 수 있는 AMBA 버스에 IP를 추가할 수 있는 인터페이스 구조를 연구하는 것이다.

참고문헌

[1] www.core-a.net, "Core-A Architecture Refere- nce Manual," 2008.

[2] www.eclipse.org

[3] www.core-a.net, "Core-B Lite Specification," 2008.

[4] www.core-a.net, "Core-A AMBA Supporting Package," 2008.

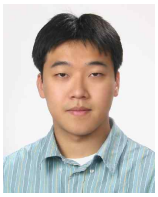
[5] www.core-a.net, "Core-A Wishbone Supporting Package," 2008.

[6] www.core-a.net, "Core-A Cache Controller Re- ference Manual," 2008.

[7] www.core-a.net, "Core-A Synthesizable Mem- ory Management Unit(MMU) Reference Man- ual," 2008.

[8] FIPS PUB 197, "The official AES standard," Nov.26, 2001

저자소개



하창수(Chang-Soo Ha)

2003년 2월 : 동의대학교 컴퓨터공학과

2006년 2월 : 동의대학교 컴퓨터·소프트웨어공학 석사

2006년 3월 ~ 현재 : 동의대학교 컴퓨터응용공학 박사과정

※관심분야 : 그래픽 프로세서 설계, 임베디드 시스템 및 SoC 설계

최병윤(Byeong-Yoon Choi)

1985년 2월 : 연세대학교 전자공학과

1992년 8월 : 연세대학교 전자공학과 공학 박사

2006년 1월 ~ 2006년12월: 오클랜드대학 방문 연구교수

1993년 3월~현재 : 동의대학교 교수

※관심분야 : RISC 마이크로프로세서 설계, 그래픽 및 암호 알고리즘의 SoC 설계