32 비트 RISC 코어의 SystemC 모델링

최 홍 미, 박 성 모
전남대학교 컴퓨터공학과
전화 : 062-363-8898 / 핸드폰 : 018-790-8898

32 Bit RISC Core modeling using SystemC

Hong-Mi Choi, Seong-Mo Park
Dept. of Computer Engineering, Chonnam National University
E-mail : hmchoi@ciscom.chonnam.ac.kr

Abstract

In this paper, we present a SystemC model of a 32-Bit RISC core which is based on the ARM7TDMI architecture. The RISC core model was first modeled in C for architecture verification and then refined down to a level that allows concurrent behavior for hardware timing using the SystemC class library. It was driven in timed functional level that uses handshake protocol. It was compiled using standard C++ compiler. The functional simulation result was verified by comparing the contents of memory, the result of execution with the result from the ARMulator of ADS(Arm Developer Suite).

I. 서론

반도체 공정 기술의 발전으로 마이크로프로세서, 메모리, 로직, 아날로그, DSP 등 다양한 기능을 가진 반도체 칩들을 하나의 칩으로 집적하는 개념의 SoC(System-On-Chip) 기술이 부각되고 있다. 그러나 기존의 시스템 설계 방법은 크고 복잡한 SoC 설계 요구를 만족시키지 못하고 있다. 하드웨어 개발이 끝나는 시점에서 소프트웨어 개발을 시작하고, 그 후에 통합, 검증의 절차를 거치는 기존의 설계 방법으로는 개발 기간을 단축시키기 어려워 제품의 경쟁력을 확보하지 못하기 때문이다. 새롭게 대두되는 시스템 설계 방법론은 보다 신속하게 제품을 개발하기 위해 재사용 가능한 IP 기술과 하드웨어와 소프트웨어 공동 설계 환경 및 통합화 기술이 중심이 되고 있다. 특히, 하드웨어와 소프트웨어 통합 개발이 가능한 시스템 레벨의 설계 방법으로 C/C++ 기반의 하드웨어 설계 툴들과 방법론들이 등장하고 있다. C/C++을 사용하여 작성된 시스템 사양은 곧바로 실행 가능한 설계 명세가 되기 때문에 전체 시스템 설계 과정에서 빠른 검증을 할 수 있게 하여 설계 기간을 단축시킬 수 있다. [1]

본 논문에서는 하드웨어와 소프트웨어 통합 개발이 가능한 대표적인 시스템 설계 언어로 SystemC를 사용하여 현재 내장형 시스템에서 널리 사용되고 있는 ARM 프로세서 코어의 명령어 셋과 호환되는 32비트 RISC 코어를 모델링 하였다. 실행 모델은 ARM 기반의 SoC 설계에서 내장형 프로세서 코어의 모델로 사용될 것이며 버스 인터페이스와 인터럽트 컨트롤러, DMA 등과 같은 주변 장치 모델과 연계하여 전체 시스템의 실행 가능한 설계 사양으로써 시스템의 제작 전에 빠른 검증과 대략적인 성능 예측을 가능하게 할 것이다. 2장에서 설계된 RISC 코어의 명령어셋과 동작 모드 등을 설명하고 3장에서 SystemC 모델링 방법과

본 논문은 한국과학재단 지정 전남대학교 고품질 전기전자부품 및 시스템 연구센터의 연구비 지원에 의해 연구되었음

모델의 실행 결과를 제시한 후 4장에서 결론을 맺었다.

II. 32 비트 RISC 코어의 특징

2.1 명령어 셋

본 논문에서 모델링한 32 비트 RISC 코어는 ARM7TDMI 구조를 기반으로 한다. 명령어 셋은 표 1에서 보여지는 것처럼 9가지로 분류된다. 데이터 처리 명령과 곱셈 명령은 산술 및 로직 연산에 관련되며, 단일 및 블럭 데이터 전송, 스왑 명령은 레지스터와 메모리 간 데이터 전달을 제어하고 PSR 전송 명령은 상태 레지스터와 일반 레지스터 간 데이터 전달에 관여한다. 분기 명령은 명령어 실행 흐름를 제어하고 소프트웨어 인터럽트 명령은 특권 모드(Priviliege mode)로 들어갈 때 사용된다. 모든 명령어가 조건적인 실행을 할 수 있는데 이는 Negative, Zero, Carry, Overflow와 같은 프로그램 상태를 나타내는 플래그 비트들에 의해 실행 여부를 결정하게 된다. 32 비트 명령어의 최상위 4 비트는 Equal, Not equal, Less나 greater than 등과 같은 15가지 타입을 정의한다.[2][3]

표 4. 32 비트 명령어 셋

분류	명령어
Data processing	AND, EOR, SUB, RSB, ADD, ADC, SBC, RSC, TST, TEQ, CMP, CMN, ORR, MOV, BIC, MVN
Multiply/Multiply Long	MUL, MLA, UMULL, UMLAL, SMULL, SMLAL
Single data transfer	LDR, STR, LDRB, STRB
Halfword data transfer	LDRSB, LDRSH, LDRH, STRH
Block data transfer	LDMIB, LDMIA, LDMDB, LDMDA, STMIB, STMIA, STMDB, STMDA
Single data swap	SWP, SWPB
PSR transfer	MRS, MSR
Branch/Exchange	B, BL, BX
Software Interrupt	SWI

주소 지정 모드는 크게 PC 상대 모드와 베이스 레지스터 상대 모드가 있고, 모든 데이터 전송 명령어는 Write Back(W) 비트에 의해 실행 과정에서 수정된 값을 자신의 베이스 레지스터에 다시 쓰거나 무시할 수 있다. 블록 데이터 전송 명령어는 16개의 연속적인 Load/Store 명령어를 하나의 명령어로 실행 가능케 하여 프로세서 상태 및 컨텍스트를 저장, 복구하거나 메모리로부터 데이터를 블록으로 전달할 수 있게 한다.

2.2 동작 모드

32 비트 RISC 코어의 동작은 정상적인 프로그램 상태인 사용자 모드, 일반적인 인터럽트 처리를 위한 IRQ(Interrupt Request) 모드, 외부적인 데이터에 대한 빠른 인터럽트 처리를 위한 FIQ(Fast Interrupt Request) 모드, 운영체제에 의해 보호되는 모드인 수퍼바이져(Supervisor) 모드, 메모리로부터 잘못된 데이터나 명령어를 가져올 때 발생되는 중단(Abort) 모드, 정의되지 않은 명령어에 대한 비정의(Underfined) 모드 등 6개의 동작 모드로 구성된다. 동작 모드들 간의 전환은 PSR 명령어에 의해 CPSR(Current Program Status Register) 레지스터 값이 수정되거나 인터럽트 발생, 예외 처리에 의해 수행된다. 레지스터 셋은 37개의 레지스터와 6개의 프로그램 상태 레지스터를 사용하고 있다. 사용자 모드에서 16개의 레지스터를 사용하며 FIQ 모드에서 빠른 인터럽트 처리를 위해 별도로 6개의 레지스터를 두고 그외 모드에서는 각각 2개씩의 레지스터를 따로 두어 스택 포인터와 링크 레지스터로 사용하고 있다.

2.3 Thumb 명령어

본 논문에서는 32 비트 ARM 명령어 셋 이외에도 ARM 명령의 압축된 형태인 16비트 Thumb 명령어를 지원한다. Thumb 명령어는 명령어 메모리로부터 가져오며 명령어 디코딩 과정에서 대응하는 32 비트 ARM 명령어로 확장된다. 프로그램 실행 시 ARM과 Thumb 명령어 모두 실행 가능하며 BX 명령어를 통해 CPSR의 State bit(T)가 세트되며 ARM/Thumb 프로세서 상태가 천이 된다.

III. RISC 코어의 SystemC 모델링

3.1 SystemC를 이용한 모델링 방법

SystemC는 Synopsys, Coware를 중심으로 구성된 OSCI(Open SystemC Initiative)에서 개발한 시스템 모델링 환경으로 C++ 클래스 라이브러리와 시뮬레이션 커널로 구성된다.[4] 이는 하드웨어 개념인 병렬 처리나 반응적 구동 특성, 타이밍, 비트/벡터의 데이터 형 등을 각각 클래스 형태로 제공하여 C/C++의 상위 레벨에서 하드웨어를 모델링 할 수 있게 한다. 또한

SystemC로 기술된 코드는 여러 EDA 벤더들이 제공하는 툴에 의해 컴파일 되어 HDL로 자동 변환 되거나 합성 및 FPGA 프로토타입 제작을 가능하게 하고 있다.[5]

본 논문에서는 SystemC에서 제공하는 3가지 유형의 프로세스 중에서 Clocked Thread 프로세스를 사용하여 클럭의 상승 에지를 이벤트로 동작하는 하드웨어를 모델링하였다. Method 프로세스와는 달리 Thread 프로세스는 while 루프를 무한대로 순환시키며, 프로세스 내에서 wait()나 wait_until() 함수들을 이용하여 신호에 이벤트가 발생할 때까지 지연시키고 신호값의 변화에 따라 다음 동작을 연속할 수 있게 한다. [6]

3.2 RISC 코어 모델링

SystemC 설계 추상화 단계는 UTF(Untimed Functional), TF(Timed Functional), BCA(Bus Cycle Accurate), CA(Cycle Accurate) 과정으로 이루어져 있으며 정제(refinement) 과정을 통해 하드웨어의 정확한 사이클 정보를 포함하는 RTL까지 기술하게 된다. 본 논문에서는 코어 구조 검증 차원에서 구체적인 타이밍 정보와 하드웨어적인 구조가 없는 UTF 단계의 C 모델을 기술한 후, 하드웨어 구조를 고려하여 모듈 단위로 나누고 델타 지연의 시간 개념을 적용하여 TF 단계에서 코어 모델을 기술하였다. 본 연구실에서 설계 개발한 RISC 코어를 참조하였고 같은 검증 환경을 따랐다.[7]

RISC 코어의 동작 단계는 크게 명령어 페치, 디코딩, 실행 과정으로 이루어진다. 페치 과정에서는 ADS의 AXD Debugger로부터 추출한 바이너리 파일이 저장된 프로그램 메모리로부터 명령어를 읽어오고 디코딩에서는 Thumb 명령어를 대응되는 ARM 명령어로 확장한 후 디코딩하고 실행에 필요한 레지스터를 선택한다. 실행 과정은 선택된 레지스터 값을 읽고 해당되는 연산을 거쳐 결과를 레지스터에 저장하거나 데이터 메모리에 저장한다. 이러한 동작들을 바탕으로 SystemC 모델이 구성되었으며 그림 1은 이를 나타내고 있다.

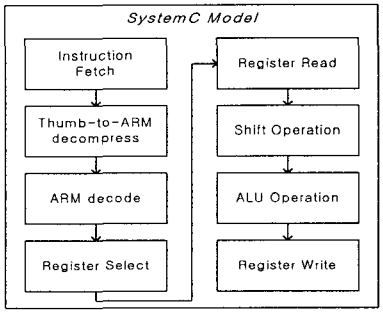

그림 1. RISC 코어의 SystemC 모델

작성된 SystemC 모델의 파일 구성은 그림 2와 같다. 프로그램 실행 초기화 과정에서 표 1의 명령어들로 구성된 바이너리 파일이 프로그램 메모리 및 데이터 메모리에 로드된다. 페치 모듈에서 cs 신호를 통해 프로그램 메모리에 접근하고 pc값으로 주소 지정을 하여 명령어를 불러온다. 명령어 데이타가 디코드 모듈에 전달되며 먼저 16 비트 Thumb 명령어를 32비트 ARM 명령어로 확장하는 decompress 과정을 거치고 명령어의 최상위 4비트가 나타내는 조건 필드를 분류하여 실행 여부를 결정한다. 명령어 실행 조건이 만족하지 않으면 pc 카운팅을 4씩 증가시켜 다음 명령어 페치로 연결되고 그렇지 않은 경우 32비트 ARM 명령의 디코딩 과정을 거쳐 실행을 하게 된다.

연산 명령은 산술 및 논리 연산을 위해 Shifter, ALU 모듈이 동작하고 곱셈 연산을 수행하는 Multiplier 모듈이 동작하게 된다. 메모리 접근은 Load/Store와 Swap 명령을 통해서 데이터를 읽고 쓴다. 메모리 읽기와 쓰기 타이밍을 위해서 mem_valid, mem_access, mem_rw 신호가 메모리 동작을 구동시킨다. 특히 가장 많은 수행 사이클을 필요로 하는 블록 데이터 전송 명령의 경우, stall_fetch 신호를 발생시켜 모든 데이터 전송이 끝날 때까지 다음 명령어 페치 동작을 지연시킨다. 메모리 접근 신호가 발생하여 데이터 메모리 모듈이 구동되면 mem_addr 신호값을 읽고 해당된 메모리 번지로부터 데이터를 읽거나 쓰기 동작을 한다. 블록 데이터 전송 명령은 이러한 일련의 과정을 레지스터 리스트에 기록된 모든 레지스터에 대해 적용되며 레지스터 R0부터 R15까지 반복한다.

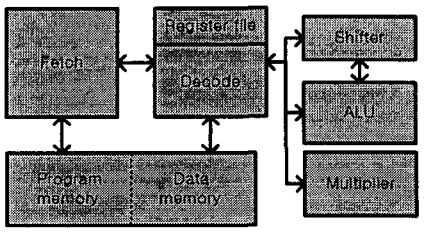

그림 2. RISC 코어 모델의 파일 구성

프로그램 카운터 계산은 PC값이 저장되는 레지스터 R15값에 의해 다음 명령어가 결정된다. 대부분의 명령어는 next_pc 신호를 통해 증가하게 되며 Branch 명령의 경우는 branch_driven 신호에 의해 branch의 타겟 주소가 다음 pc값으로 결정된다. 이밖에 R15에 값

을 직접 입력시켜 분기 명령과 같은 결과를 보여주는 명령어로 목적지 레지스터가 R15인 ADD, SUB 등의 데이터 처리 연산 명령과 단일 데이터 전송 명령인 LDR, 레지스터 리스트의 15번째 비트가 1로 체크된 블록 데이터 전송 명령어가 있다. 이러한 명령어들에 대해서는 stall_fetch 신호를 발생시켜 다음 명령어 페치를 지연시키고 처리된 값이 R15에 저장한 후 다음 명령을 실행시킨다.

3.3 모델의 실행 결과

본 논문에서는 32 비트 RISC 코어를 SystemC를 이용하여 행위 수준에서 모델링하였으며, 검증을 위해 그림 3과 같은 디버깅 환경을 이용하였다. C언어로 기술된 응용 프로그램들을 트랩 처리기, ROM/RAM 매핑 및 각 모드별 스택 포인터 초기화 프로그램들을 포함한 환경에서 ADS의 ARM 컴파일러 및 어셈블러, 링커를 사용하여 ARM 명령어로 컴파일된 결과를 바이너리 파일로 추출하여 SystemC 모델의 프로그램 메모리와 데이터 메모리에 각각 로드하여 프로그램을 실행시켜 보았다. 그림 3에서 SystemC 모델의 실행 결과들 중 레지스터 파일 내 각 모드별 레지스터 값과 CPSR과 모드별 SPSR 값, 메모리 접근 주소와 데이터, 프로그램 카운터 값들을 ADS에서 제공하는 코어 시뮬레이터인 ARMulator에서 응용 프로그램 실행 결과와 각 명령어 별로 비교하여 일치하는 것을 확인하였다.

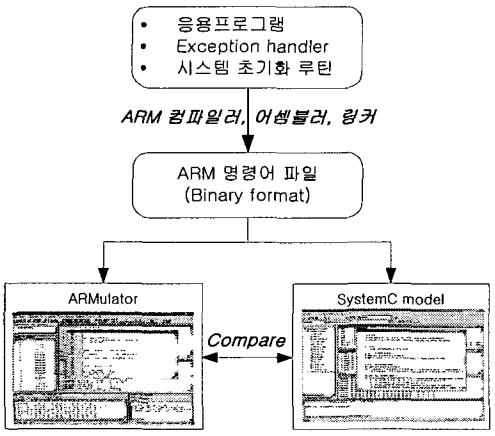

그림 3. 디버깅 환경

그림 4는 코어 모델의 실행 결과로 각 명령어에 따른 레지스터 값의 변화를 보여주는 파일과 결과 데이터 메모리, 신호값의 변화에 따른 VCD 출력파형 등을 타나낸다.

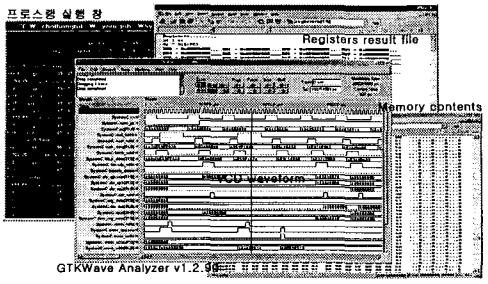

그림 4. 코어 모델의 실행결과

IV. 결론

본 논문에서는 하드웨어와 소프트웨어 통합 설계 및 개발 환경의 중요성이 증대되면서 등장한 SystemC 설계 방법을 적용하여 32비트 RISC 코어를 모델링 하고 ADS를 통해 동작을 확인하였다. 일반적인 C 모델에 비해 보다 더 하드웨어적인 개념이 적용되었고 HDL로 기술한 RTL 수준에는 미치지 못하지만 행위 수준에서 동작하며 각 프로세스는 병렬처리 된다. SystemC의 설계 추상화 수준에서 살펴보면 Timed Functional 레벨에서 기술되어 델타 지연을 이용하여 대략적인 타이밍을 보여줄 수 있다.

설계된 코어 모델은 특정한 버스 인터페이스 및 주변 장치들과 연계하여 전체 시스템의 제작에 들어가기 전에 빠른 검증과 대략적인 성능 예측을 할 수 있는 실행가능한 설계 모델로서 활용될 것이다. 앞으로 더 정확한 검증을 거치고 외부 인터페이스에 대한 정의를 명확하게 하여 SoC를 위한 IP로서 이용할 계획이다.

참고문헌

[1] Giovanni De Micheli, "Hardware Synthesis from C/C++ Models", Proceedings of the DATE'99 conference, pp.382-383, March 1999

[2] ARM Architecture Reference, Advanced RISC Machines. Ltd., Cambridge, U.K., 1995

[3] S. Furber, ARM System Architecture, AW, 1996

[4] SystemC http://www.systemc.org

[5] X. Liao S.Tjiang and R. Gupta, "An Efficient Implementation of Reactivity for Modeling Hardware in the SCENIC Design Environment", Proceeding for the DAC'97, pp.70-75, June 1997

[6] SystemC, Version 2.0, User's Guide, 2001

[7] 정갑천, 박성모, "휴대 단말기용 32 비트 RISC 코어 구현", 대한전자공학회 논문집, 제38권 CI편, 제6호, pp. 82-92, 2001.11