내장형 제어용 프로세서를 위한 명령어 기반 범용 시뮬레이터 개발

양훈모, 정종철, 김도집, 이문기
연세대학교 전기전자공학과 VLSI&CAD 연구실
전화 : 02-2123-4731 / 핸드폰 : 016-294-9173

A Design of Instruction-Set Based Simulator of Processor for Embedded Application System

Hoon Mo Yang, Jong Cheol Chung, Do Jip Kim, Moon Key Lee
Dept. of Electric and Electronic, Yonsei University
E-mail : slover@spark.yonsei.ac.kr

Abstract

As SOC design methodology becomes popular, processors, the essential core in embedded system are required to be designed fast and supported to customers with expansive behavior description. This paper presents new methodology to meet such goals with designer configurable instruction set simulator for processors. This paper proposes new language called PML(Processor Modeling Language), which is based on microprogramming scheme and is also successful in most behavior of processors. By using this, we can describe scalar processor very efficiently with by-far faster simulation speed in compared with HDL model.

I. 서론

근래 SOC 설계 기법이 보편화되면서 IP의 중요성이 부각되고 있다. 따라서 SOC 기법의 주 적용 분야인 내장형 응용 제어 시스템의 경우, 그 핵심을 이루는 프로세서에 대해, 첫째 Time-to-market을 극소화하기 위하여 아키텍처에 대한 동작 수준 검증 시간을 단축시켜야 하며, 둘째, 설계된 IP를 유통시킬 때, 프로세서 동작의 세부적인 사항을 쉽게 검증 확인 할 수 있는 환경을 제공할 의무가 있다.

이러한 2가지 조건을 만족시키기 위해, 프로세서 설계자는 해당 아키텍처에 대한 동작을 기술한 모델을 HDL이나 C/C++ 등으로 작성한 전용 시뮬레이터의 형태로 제공하나 HDL의 경우, 가장 정확한 동작 정보를 제공하는 반면, 대단위 응용 프로그램을 워크로드(Workload)로 사용할 경우, 속도와 용량에서 제약을 받으며 범용 언어를 이용하여 작성한 시뮬레이터는 하드웨어의 병렬성을 기술하는데 제약을 지니고 프로세서의 종류에 따라 일관된 형식을 지니지 못한다는 단점이 존재한다.

따라서 본 논문은 HDL보다 속도 면에서 더 효율적이며 동시에 범용 언어보다 더 편리하게 프로세서 동작을 기술할 수 있는 새로운 기법을 제안한다.

PML(Processor Modeling Language)로 명명한 다음과 같은 사양을 지원한다.

첫째, 분기 명령어 처리, 예외 처리, 레지스터 bypass 및 레지스터 메모리 locking 등 파이프라인 구조를 지닌 스칼라 프로세서와 관련된 동작을 유연하게 기술한다.

둘째, 뱅크 구조 레지스터, 레지스터 윈도우 등의 다양한 형태의 레지스터 파일 구조에 대한 동작 기술이 가능하다.

셋째, 임의로 파이프라인을 설정할 수 있으며 각각의 경우에 대하여 클락 사이클 단위의 타이밍이 기술 가능하며, 이러한 구조는 top-down 방식으로 기술되어 모델에 대한 분석이 더욱 명료해 진다.

넷째, 프로세서 시뮬레이터를 설계함에 있어서 가장

오류가 발생하기 쉽고 번잡한 작업인 명령어 디코더의 구현부를 자동으로 생성함으로써 시뮬레이터 설계 시, 편이성을 높였다.
본 PML은 C++로 작성하였으며, 인터프리터 방식 대신, 설계자가 제공하는 소스 파일을 파싱하여 라이브러리를 생성하고 이를 컴파일하여 구축하는 방식을 채택함으로써 전체적인 수행 속도를 높일 수 있었다.

II. PML 기반 프로세서 모델 설계

2.1 기본 구조

PML 기반 프로세서 모델의 전체 블록도는 그림 1과 같다. 기본적으로 PML 모델은 마이크로프로그래밍 방식으로 구동한다. Sequencer 블록은 내부적으로 정의한 마이크로명령어가 저장되어 있으며 페치된 새로운 매크로명령어는 디코드 테이블에 입력으로 제공되고 이를 통해서 Sequencer 블록 상 진입 번지를 구한다. 진입 번지 이후에 수행될 마이크로명령어는 내부 마이크로 분기 명령어에 의해 흐름 제어를 수행하며 해당 매크로 명령어에 대한 연산을 수행한다.

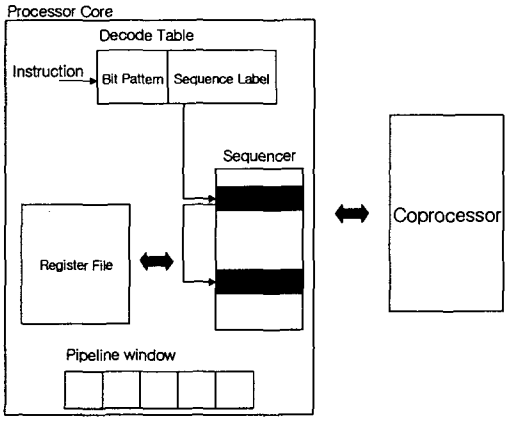


그림 1 PML 기반 프로세서 모델의 전체 블록도

Pipeline window 블록은 수행되는 마이크로명령어와 연관을 지니며 파이프라인 상태 및 타이밍을 관장한다. 프로세서 모델은 하나의 코어와 다수 개의 코프로세서로 구성된다. 또한 각각의 프로세서는 그림 1의 코어와 동일한 구조를 지니며 특정 마이크로명령어에 의해 이들간 흐름 분기가 수행되어 전체적으로 동기를 맞추며 수행되는 구조를 이룬다.

2.2 명령어 디코더 모델

본 시뮬레이터는 해시 테이블을 이용하여 디코딩을 수행함으로써 임의의 비트 필드에 대해 평균적인 수행 속도를 보이는 디코더 모델을 구축하였다.
디코더 모델의 해시 테이블 구조는 그림 2와 같다. 사용자는 총 명령어 비트의 수와 최대 탐색 수를 명시한다. 그림 2의 경우, 16비트 명령어에 대해 4비트씩 최대 탐색 4번을 수행하는 경우이다. 각각의 해시 노드에는 추출할 비트열에 대한 첫 번째와 마지막 위치에 대한 정보가 저장되어 있고 그 비트를 그대로 해쉬 노드의 인덱스로 사용하여 접근한다. 만약 해당 엔트리의 다음 포인터가 NULL 값이면 디코딩이 완료된 것이고 아니면 계속 수행한다.
명령어 디코더는 수행할 명령어의 형태를 결정하는 작업이기 때문에 명령어의 피연산자를 구분하는 비트열과는 무관하다. 따라서 이러한 비트열은 don't care로 처리함으로써 전체적으로 탐색 속도를 높이는 구조로 이루어져 있다.
이러한 디코더 모델은 사용자가 명령어 형태를 임의로 구분하고 비트 패턴만 명시해 주면 자동적으로 생성하게 함으로써 시뮬레이터 구현의 편이성을 증진하였다.

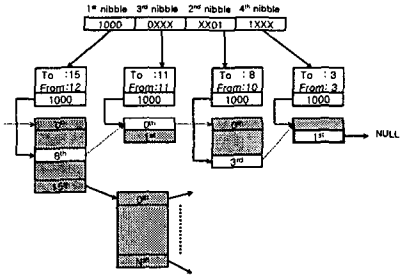


그림 2 해쉬 테이블을 이용한 명령어 디코더 모델

2.2 레지스터 파일

본 시뮬레이터는 프로그램 접근 가능한 레지스터뿐만 아니라 내부 구조상 사용되는 하드웨어 래치 및 명령어 상 피연산자를 나타내는 비트열, 내부 마이크로명령어의 흐름 제어를 위한 정보를 포함하여, 시뮬레이터 내부적으로 이용 가능한 모든 데이터를 레지스터로 취급한다. 이와 같은 방식은 전체적으로 PML을 기술하기가 매우 간편해지고 보다 다양한 아키텍처에 대한 내부 구조의 유연한 기술이 가능해진다는 장점이 있다.
본 시뮬레이터는 C++의 가상 함수를 이용하여 이러한 다양한 피연산자에 대한 접근을 읽기를 위한 GetReg 함수와 쓰기를 위한 PutReg 함수만을 이용한 단일화된 인터페이스를 제공한다. 또한 사용자는 이러한 함

수를 직접 호출하는 것이 아니라 프로세서의 동작을 기술한 마이크로프로그램 상 인덱스를 표기함으로써 간접적으로 이루어진다.

그림 3은 레지스터 정의부의 예를 나타낸다. 첫 번째 부분은 'CPSR 레지스터의 값을 DELTA_BASE와 더한 인덱스 값을 참조하여 레지스터를 엑세스한다. 따라서 읽는 값은 상수 4 또는 2가 된다.

두 번째 부분은 피연산자 디코딩에 적용할 수 있는 예를 보여준다. 'IR_RN은 명령어를 저장한 'IR 레지스터 중 1-0 비트를 추출한 값을 제공한다. 이후 'RN 레지스터는 'IR_RN값과 GENERAL이란 값을 더한 인덱스를 가진 위치의 레지스터 값을 제공한다. 따라서 'IR_RN 값에 의해 4개의 레지스터 중 하나를 선택하게 된다.

세 번째 부분은 특정 모드에 의해 레지스터의 일부분이 갱신되는 경우를 구현한 예이다. 'CPSR의 값이 참이면 마스크 0x00ff가 적용되어서 'PROTECTED_REAL 레지스터 값은 일부 비트가 쓰기 금지된다.

마지막 경우는 뱅크 구조 레지스터를 구현하기 적합한 예로서 'Mode가 'CPSR의 값에 따라 'USR, 'UND 등으로 정의되어 있다. 따라서 R0나 R1을 접근할 때, 해당 모드에 의해 서로 다른 레지스터 값이 선택된다.

```
// Base+Index
DELTA: @DELTA_BASE + <CPSR>;
DELTA_BASE: Const 4;
            Const 2;

// Operand decoding
IR_RN: IR[1:0];
RN: @GENERAL + <IR_RN>
GENERAL: Var;
         Var;
         Var;
         Var;

// Mask
PROTECTED: CPSR ? 0x00ff @PROTECTED_REAL
PROTECTED_REAL: Var;

// Mode
MODE  CPSR {
  USR : 0x01;
  UND : 0x02;
          .....
};

R0 : USR @R0_USR, UND @R0_UND, .....;
R1 : USR @R1_USR, UND @R1_UND, .....;
```

그림 3. 레지스터 정의

이와 같이 사용자는 레지스터 정의 구역을 직접 작성함으로써 대부분 현존하는 레지스터 파일 구조를 기술할 수 있게 된다.

2.3 마이크로프로그래밍

시뮬레이터의 제어부는 PML을 이용하여 사용자가 기술하는 형식으로 되어 있으며 자체 컴파일러를 이용하여 이를 라이브러리화 하여 시뮬레이터에 링크시켜 구동하는 형식으로 구성하였다.

제어부 기술은 파이프라인 단을 기본 구성 단위로 삼은 마이크로프로그래밍 방식으로 기술된다. 개개의 마이크로명령어의 기본 형식은 다음과 같다.

```
+ <소요시간> <명령어> <레지스터 목록>;
[예제 1] // R0:=R1+R2 소요 시간 1사이클
         +1 ADD R0, R1, R2;
```

명령어는 C++의 가상 함수를 이용하여 작성하였으며 사용자의 필요에 따라 확장이 가능하도록 구성하였다.

마이크로프로그램은 내부적으로 흐름 제어가 가능하다. 흐름 제어 문구는 C 언어와 유사하게 조건 분기문인 if와 루프문인 for, while 등을 지원한다.

```
[예제 2] while( eqt COUNT) {
          +1 ADD R0, R0, 1;
          SUB COUNT, 1;
         }
```

예제 2는 레지스터 'COUNT 값이 0이 아닐 경우, R0의 값을 하나씩 증가시키는 구문을 나타낸다.

본 시뮬레이터는 분기 조건문으로서 매크로명령어 문맥에 따른 특수한 조건문을 제공한다. 예제 3과 4는 각각의 예를 나타낸다.

```
[예제 3] switch($) {
          case [ADD]: ADD R0, R1, R2; break;
          case [SUB]: SUB R0, R1, R2; break;
          .....
         }
[예제 4] switch(%IR[3:2]) {
          case [0x00] : ADD R0, R1, R2; break;
          case [0x01] : SUB R0, R1, R2; break;
          ....
         }
```

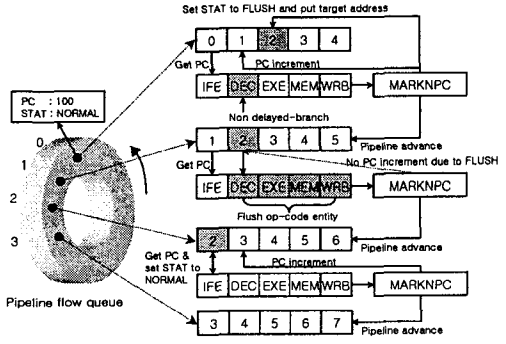
예제 3의 경우, $는 현재 수행 중인 매크로명령어의 식별자를 의미한다. 따라서 식별자에 따라 각각 분기가 발생한다. 시뮬레이터는 각각의 식별자에 대한 분기 테이블을 구성하고 있다.

예제 4의 %'register는 레지스터의 부분 비트 열의 값에 따라 분기를 수행한다. $와는 달리 이 경우엔 'Base+'Index 방식을 이용하여 레지스터 정의 부에 컴

파일러가 별도로 정의한다.
매크로명령어의 형태가 비트 필드와 일관성이 없을 경우에는 $ 조건문을, 있을 경우에는 %를 적용하면 편리하다.
명령어 형태를 디코더 해시 테이블에 의해 이미 세분화하였을 경우에는 $와 %를 사용할 필요가 없으며 속도 면에서 유리하다. 반면에 특정 프로세서의 명령어 세트에 대해서는 디코더 해시 테이블만을 이용하여 구분하는 것이 용량 면이나 유지 보수 면에서 매우 비효율적일 수 있다. 예를 들어서 ARM 프로세서의 경우, 시프트 연산과 ALU 연산이 하나의 명령어 세트에 융합되어 있는데 이 경우, 해시 테이블 수준에서 분류할 경우, 테이블의 크기가 매우 커지게 된다. 따라서 이러한 경우에는 $와 % 조건문을 사용하는 것이 훨씬 효율적이다.

2.4 파이프라인 제어 모델링

본 시뮬레이터는 그림 와 같은 구조의 환형 큐를 이용하여 파이프라인 흐름을 제어한다. 하나의 매크로 명령어가 수행 종료될 때마다 MarkNpc 서브루틴에 의해 각각의 파이프라인 슬롯에 분기할 주소를 기록하며 매 페치 때마다 이 주소를 참조하여 다음 명령어를 가져온다. 예외나 분기명령어에 의한 플러쉬가 발생하면 해당 슬롯에 분기 주소와 함께, 플러쉬 상태를 명기하며 매 마이크로명령어는 이 상태 정보를 이용하여 플러쉬를 감지할 경우, 수행 중인 명령어를 바로 종료시키는 구조로 이루어졌다. 그림 은 분기 명령어에 의해 1개의 명령어가 플러시 되는 과정을 예시한다.



2.4 파이프라인 제어 모델링

개개의 파이프라인 단은 고유의 시간 변수가 할당되어 있으며 각각 현재 수행된 매크로 명령어에 대해 해당 파이프라인 단의 점유 초기 시점과 종료 시점을 기록하고 있다. 시간 변수의 변화는 개개 마이크로명령어의 수행 소요 시간과 파이프라인 단 간의 동기를 맞추기 위한 파이프라인 스톨 시간의 합에 의해 결정된다. 실제 파이프라인 구조는 모든 파이프라인 단이 중첩되면서 한 번에 여러 개의 명령어에 대한 부분 동작을 동시에 수행해 나간다. 반면 본 시뮬레이터는 순차적으로 명령어 하나씩 처리하는 구조로 이루어져 있다. 이와 같은 방식은 임의의 파이프라인 단에서 변하는 가능성이 있는 상태 정보를 포함할 경우, 완벽한 동작 기술이 불가능하다는 단점이 있으나, 대부분의 프로세서의 경우, 그러한 경우는 극히 드물며, 프로그램 구조가 전체적으로 단순해지는 장점이 있다.

결 론

본 논문에서 제안한 시뮬레이션 기법은 모델링 언어를 사용하여 범용적으로 프로세서를 기술하여 시뮬레이션 할 수 있게 함으로써 설계자의 요구를 충족시킬 수 있는 사양의 시뮬레이터를 간편하게 설계할 수 있는 방법을 제공하였으며 ARM7 아키텍처에 대한 모델을 테스트한 결과, HDL 모델에 비해 200배정도 빠른 속도를 보였다. 추후 연구 과제로는 기존의 스칼라 방식에서 수퍼스칼라 방식으로의 확장을 들 수 있다.

참고문헌

[1] B.Y.Choi, K.Y.Lee, S.H.Lee, and M.K.Lee, VLSI design of a pipeline Controller for a 32-bit Application-Specific RISC , in KITE Journal of Electronic Engineering, vol. 5, no. 2 Dec 1994

[2] Ron Cates, Processor Architecture Considerations for Embedded Controller Applications , in IEEE MICRO, 1988

[3] Moon Gyung Kim, Byung In Moon, Sang Jun An, Dong Ryul Ryu, and Yong Surk Lee, Implementation of a cycle based simulator for the design of a processor core , the 1st IEEE Asia Pacific Conference on ASICS, Seoul, Korea, Aug. 1999