64 Bit EISC 프로세서 설계

임 종윤(林 種 潤)
이 근 택(李 根 澤)
아시아 디자인 PLI
광운 대학교 지능정보 처리 공학과
전화 : (02) 545-4484 / 팩스 : (02) 545-4485

64 Bit EISC Processor Design

Jong Yoon Im , Geun Taek,Lee
Dept of intellectual-information , Kang-Woon Universjty , Asia design corp
E-mail : sazabi@chollian.net, gtlee@adc.co.kr

Abstract

The architecture of microprocessor for a embedded system should be one that can perform more tasks with fewer instruction codes. The machine codes that high-level language compiler produces are mainly composed of specific ones, and codes that have small size are more frequently used. Extended Instruction Set Architecture (EISC) was proposed for that reason. We have designed pipe-line system for 64 bit EISC microprocessor. function level simulator was made for verification of design, and instruction set architecture was also verified by that simulator. The behavioral function of synthesized logic was verified by comparison with the results of cycle-based simulator.

I. 서론

1970년대에 개발된 마이크로 프로세서는 프로그램을 이용한 제어 방법을 채택함으로서 시스템 개발의 용이성과 우수한 성능을 바탕으로 거의 모든 전제 제품 및 자동화 제어 기기에서 채용하고 있다.

초기의 제어용 마이크로 프로세서는 4비트와 8비트가 주류를 이루었으나 1990년대 후반부터는 16비트와 32비트 프로세서가 주도하고 있으며 향후 시장에서는 64비트 시장이 형성되어 주도될 것으로 전망된다.

제어 실장용 마이크로 프로세서는 명령어의 밀도(Code Density)가 높아야 하며 동작 속도가 빨라야 하고 구조가 간단하여야 경쟁력이 있는 가격과 성능을 보장받을 수 있다. 이러한 요구에 부응하기 위하여 각 반도체 개발 회사는 경쟁력 있는 제어 실장용 마이크로 프로세서의 개발에 박차를 가하고 있다.

현재의 설계상의 흐름은 상위 언어의 컴파일러가 생성해 내는 명령어들의 분포 특성을 명령어 구조에 반영하여 ISA(Instruction Set Architecture)를 만들고 있다. C 언어와 같은 상위 수준 언어의 컴파일러가 만들어 내는 명령어들은 특정 명령어와 일정 크기 이하의 상수 값을 사용하는 경우가 그 외의 명령어 혹은 일정 크기 이상의 상수를 사용하는 경우보다 높게 나온다. 즉 같은 동작을 하는 명령어라 하여도 16비트 크기 이하의 변위(offset)를 사용하는 명령어의 빈도가 17 비트부터 32비트 크기의 변위를 사용하는 경우보다 훨씬 높게 나온다. 이렇게 상황에 따라 틀린 변위를 고려하지 않고 일정 크기로 고정되어 있는 변위를 사용하는 명령어 구조는 명령어 구성상에 필요 이상의 과도한 명령어 혹은 Operand Field를 요구하게 되므로 적합하지 않다. 따라서 상황에 맞게 적절하게 확장하여 변위를 생성할 수 있는 확장 명령어 구조를 가진 EISA

(Extended Instruction Set Architecture) 구조가 적합하다.

본 논문에서는 확장 명령어 구조를 가지는 64 비트 EISC (Extended Instruction Set Computer) 프로세서인 AE64000을 기능 수준의 검증과 Cycle Based Simulator를 사용하여 기능을 검증하고 HDL을 사용하여 합성 가능한 수준으로 설계하는 것을 보이도록 하겠다.

II. 확장 명령어 구조 (Extension Instruction Set Architecture)

상위 수준 언어를 컴파일 할 경우 일반 연산에서 필요한 상수 값을 설정하기 위한 비트의 수는 적은 수의 비트가 요구되는 빈도가 많은 수의 비트가 요구되는 빈도 보다 훨씬 높게 나타난다.

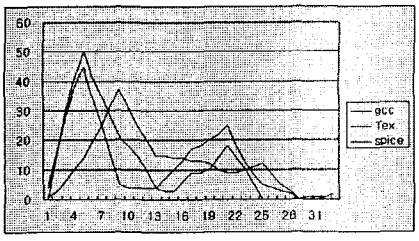

그림 1 상수 값의 표현에 필요한 비트 크기 분포[3]

그림 1에서 보는 바와 같이 상수를 표현할 때에 12 비트 이하의 상수 값에서 많은 상수를 표현할 수 있으며 23 비트 이하면 80%이상의 상수를 표현 할 수 있다.

또한 그림 2에서 보는 바와 같이 메모리의 값을 읽어들이거나 값을 기록하기 위해 필요한 변위 역시 12 비트 크기 이하로 대부분의 변위를 표현할 수 있다.

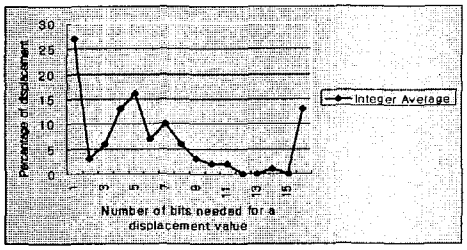

그림 2. 메모리 참조 시에 필요한 변위 값의 비트 크기 분포[3]

따라서 각각의 상황에서 요구되는 변위 또는 상수의 크기는 적을수록 그 출현 빈도가 상대적으로 높다는 것을 알 수 있다. 이러한 것을 기반으로 확장 명령어 구조는 Operand 혹은 변위를 확장시키는 개념을 도입하였다. 따라서 필요한 상수 혹은 변위를 만들 때에 필요한 크기만큼의 명령어만을 사용함으로서 높은 코드 밀도를 얻을 수 있다.

확장명령어를 지원하기 위하여 프로세서 내부에는 확장 레지스터를 두고 이를 사용한다. 확장 명령어는 List 1과 같은 알고리듬을 이용하여 확장 명령어를 수행한다.

```
if Fetched OP Code == Extension Opcode then
  if Extension Flag == 1 then
      Extension Reg = (Extension Reg) << 12 |
                      Extension Data ;
  else
  {
      Extension Reg = Sign Extend 12 Bit(
                          Extension Data
                      ) ;
      Extension Flag = 1 ;
  }
  end if
else
  Excute Operation with Extension Register ;
  if Extension Flag == '1' then
        Extension Flag = 0 ;
  end if
end if
```

표 1. 확장 명령어 수행 알고리즘

이러한 확장 명령어를 사용 할 경우 현재 확장 명령어가 수행되었다는 것을 인식하여야 다음에 수행할 비 확장 명령어가 수행될 때에 명령어 수행에 필요한 상수 혹은 변위를 확장하여 수행할 수 있다. 이를 위하여 확장 플래그를 상태 레지스터에 추가한다.

이러한 구조를 가지는 확장 명령어 구조는 기존의 구조에 비하여 같은 동작을 하기 위해서 적은 수의 명령어만으로도 원하는 동작을 구현 할 수 있다.

III. 확장 명령어를 가지는 64비트 프로세서의 파이프라인

확장 명령어를 가지는 64비트 Embedded Mirco processor 인 AE64000은 16개의 General purpose register를 가지고 있다. 또한 6개의 Special Purpose register를 가지고 있다.

GPR 0	GPR 8	USP (User Stack Pointer)
GPR 1	GPR 9	SSP (Super Stack Pointer)
GPR 2	GPR 10	Program Counter
GPR 3	GPR 11	Extension Register
GPR 4	GPR 12	Link Register
GPR 5	GPR 13	Status Register
GPR 6	GPR 14	
GPR 7	GPR 15	

그림 3. AE64000의 레지스터 세트(register set)

일반 목적의 레지스터는 여타 다른 프로세서보다 적지만 gcc (Gnu C Compiler)를 이용하여 레지스터 수의 변화에 따른 프로그램의 크기 변화를 추적하여 16개의 일반목적의 레지스터를 가질 때가 적합함을 확인하였다. [1]

Pipeline Model

AE64000은 5 Stage로 구성되어 있다.

IF	ID/OF	EXE	MEM	WB

그림 3 AE64000의 pipeline stage

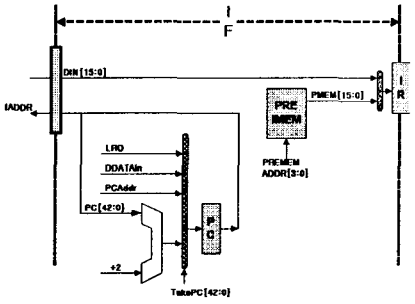

그림 4 instruction fetch Stage

Exception 처리 상황에서 processor는 PC, SR 레지스터를 stack 영역에 저장하고 Exception vector를 읽어오는 동작을 한다. 또한 Exception을 수행 후 복귀할 경우에는 위의 동작을 역순으로 수행하게 된다. 하지만 이러한 연산을 State machine에서 수행하게 되면 processor의 속도에 영향을 받는다. State machine의 속도가 전체 pipeline의 속도를 결정하는 요소가 되므로 최대한 state machine을 단순하게 설계해야 한다. 따라서 이러한 동작을 속도가 저하되는 state machine에서 수행하지 않고 해당 동작을 PREIMEM (pre-implemented instruction memory)에 미리 명령어를 저장하고 차례로 수행시킴으로서 state machine이 복잡해지지 않도록 하여 전체적으로 Pipeline의 고속 동작 가능하도록 하였다.

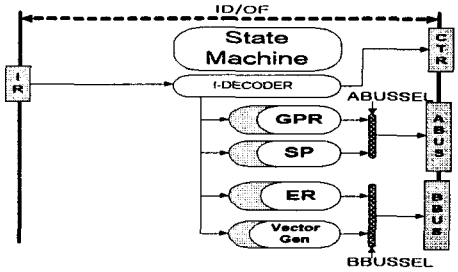

그림 5 명령어 해석 및 오퍼랜드 패치 단계

읽어온 명령어를 instruction decoder에서 decoding 하여 ABUS 혹은 BBUS에 입력될 operand와 execute , memory , write back stage에서 사용할 control code를 생성시킨다.

또한 Exception, Interrupt등이 발생하였을 때에 지정된 vector를 PC(program counter)로 불러오게 된다. 이를 처리하기 위하여 vector generation register를 두고 처리하게 된다.

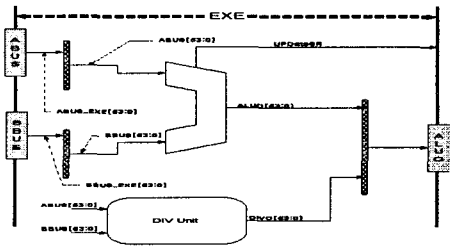

그림 6 execute stage

memory stage는 memory의 속성상 그 결과가 거의 pipeline의 critical path상에 존재한다. 따라서 해당 메모리를 읽어오기 위해서 memory stage는 연산 및 데이터 패스를 두지 않고 write back stage에서 해결하였다.

write back stage에서는 byte indexing에 따른 정렬과 함께 General purpose register에 쓰기 동작을 수행한다.

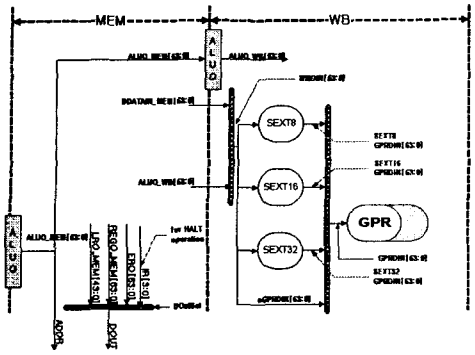

그림 7 memory stage and write back stage

Ⅳ. 확장 명령어 구조를 가지는 64비트 Processor의 검증

실제 Micro Process의 설계에서 가장 중요한 기능은 예외상황에 대한 처리 기능이다. 이러한 기능은 수시로 발생되는 예외 상황과 그 것을 처리하기 위하여 수많은 예외상황을 발생시킬 수 있는 모델을 만들고 파이프라인을 검증하여야 한다. 이를 위하여 AE64000을 설계시에 2가지 모델의 시뮬레이터를 구현하였다. 두가지 시뮬레이터는 모두 일반 C언어를 사용하여 text interface를 가지는 형태로 구현하였으며 Linux 및 Sun상에서 동작을 시켰다.

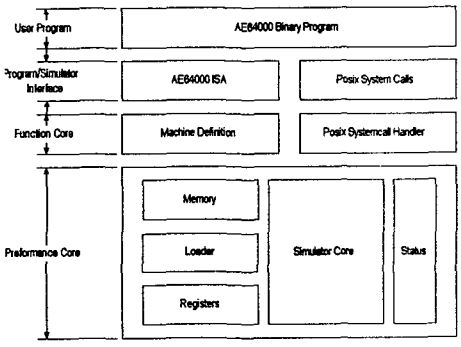

그림 8 AE64000의 시뮬레이터 구조

시뮬레이터는 구현 시에 메모리와 명령어 수행 시뮬레이터(Instruction Set Simulator)를 분리하여서 추후 메모리 관리 기 (memory management unit)과 같은 새로운 모듈을 구현 시에 용이하게 확장 할 수 있도록 구현 하였다.

Cycle based simulator는 Pipeline상에서 매 Cycle 마다 동작하는 각 스테이지의 동작을 검증하고 예외상황에서 처리되는 방식을 검증하기 위하기 위하여 구현하였다.

각각의 시뮬레이터는 내부에 불규칙하게 예외 상황을 발생 시키는 부분이 내장되어 있어 모든 상황에서 예외 상황의 수행 부분에 대한 오류를 검증하였다.

각 Cycle마다 내부 상태를 파일로 출력 할 수 있으며 출력된 결과가 실제 HDL로 움직인 결과와 동일한 지 여부를 비교하여 여러 가지 예외 상황에서도 파이프라인이 정확하게 수행하는지 여부와 HDL 모델의 동작에 대한 검증을 수행하였다.

Ⅴ. 결론 및 추후 연구

본 논문에서는 C와 같은 상위 수준의 언어가 생성하는 명령어 구조에 적합한 확장 명령어를 가진 CPU를 설계하였다. 실제 구현하기 전에 Function Level 시뮬레이터와 Cycle based 시뮬레이터를 통하여 각각 Instruction Set Architecture 및 예외 처리 상황에서의 Pipeline의 안정성을 검증하였다. VHDL로 설계하였으며 명령어를 수행 후 그 결과를 시뮬레이터와 비교하여 검증했다. 향후에는 PLI와 같은 co-verification 환경을 도입하여 VHDL에 의한 하드웨어 모델과 C 언어에 의한 소프트웨어 모델을 동시에 진행하며 검증할 것이다.

참 고 문 헌

[1] 조 경연, "확장 명령어 32비트 마이크로 프로세서에 관한 연구" 전자 공학회 논문지 제 36권 D편 제 5호 pp391-400 , May

[2] John L. Hennessy , David A. Patterson "Computer Organization & Design" 2nd Edition, M.K. Pub , 1998

[3] John L. Hennessy , David A. Patterson "Computer architecture: A Quantitative approch" 2nd Edition M.K. Pub , 1996

[4] Amos R. Omondi "The Micro-architecture of pipelined and Superscalar Computers" KAP, 1999

[5] Matthias Bauer, Wolfgang Ecker, "Hardware/Software CO-Simulation in a VHDL based Test bench Approach", Proceedings of the 34th ACM/IEEE Design Automation conference, Pages 774-779