High Performance Embedded Processor with Multiple Register Sets and Hardware Context Manager

Jeong-Ho Han[1] , Ji-Hoon Kim [1] and In-Cheol Park[1]
[1] Department of Electrical and Electronic Engineering, KAIST
jhhan@ics.kaist.ac.kr

Abstract – *As most embedded processors process multiple tasks simultaneously, multitasking effect is an important issue to be considered in designing embedded processors. Conventional multi-tasking being processed in operating systems, takes a large overhead to do context changing because operating systems must save and restore register values and decide the next task. This paper presents a new method to reduce the context changing overhead, which is associated with multiple register sets and a hardware context manager. The context switching overhead is significantly reduced, as many register values of the current context are not saved and the scheduling is performed with the hardware context manager.*

Keywords: multithread, embedded processor, multiple-register sets, hardware context manager

1 Introduction

현재 멀티미디어, 이동통신 분야, 네트워크를 비롯한 다양한 분야에서 임베디드 프로세서가 사용되고 있는데, 대부분의 경우 임베디드 프로세서는 동시에 다양한 작업을 수행하게 된다. 따라서 프로세서를 설계할 때 멀티태스킹(Multitasking)이 반드시 고려되어야 한다. 그동안 멀티태스킹은 운영체제(Operating System)에만 의존하고 있었다. 운영체제에만 의존해서 멀티태스킹을 구현할 경우, 작업전환(context switch)을 하기 위해서 프로세서의 레지스터 값들을 메모리에 저장하고 불러오는 등의 많은 부가적인 작업들이 필요하게 된다. 이러한 부가작업을 수행하기 위해 보통 수백 사이클이 필요하게 되기 때문에 실시간 시스템에서 내부 또는 외부의 요청으로 새로운 작업 수행이 필요할 때, 수백 nsec 정도의 지연 시간이 발생하게 된다. 실시간 시스템에서는 이러한 지연 시간이 중요한 요소가 되기 때문에 기존의 임베디드 프로세서를 이용한 실시간 시스템에서는 빠른 반응시간을 얻기 위해서 높은 클럭 주파수를 사용하였다. 이 논문에서는 실시간 운영체제에서 멀티쓰레딩을 효율적으로 처리하기 위해서, 여러 개의 레지스터 집합(register set)을 가지는 아키텍처를 개발하였다. 또한 운영체제에서 담당하던 스케쥴링 작업을 보조연산장치(co-processor) 형태의 하드웨어로 구현하였다. 이 두 방법을 사용해서 실시간 시스템에서 높은 클럭을 사용하지 않더라도 빠른 반응시간을 얻을 수 있으며, 이로 인해서 성능 향상뿐만 아니라 부가적으로 저전력의 특성 또한 얻을 수 있게 된다.

2 Proposed Architecture

2.1 Architecture Overview

제안된 Multiple Register Set 과 Hardware Context Manager 두 방식을 모두 적용해서 설계한 프로세서의 블록 다이어그램은 그림 1 과 같다.

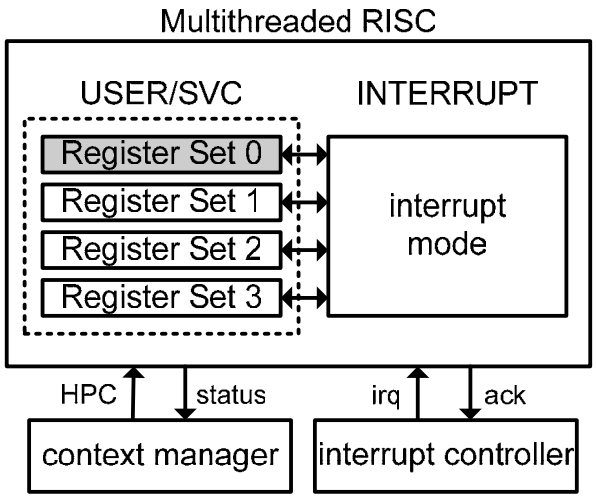

그림 1. 프로세서 블록 다이어그램

프로세서 코어 내부에는 현재 mode 에 따라서 사용할 수 있는 여러 레지스터 집합이 존재한다. 일반적인 유저모드(User mode) 또는 관리자모드 (Supervisor mode) 에서는 Resister Set0 ~ Resister Set3 을 사용하게 되며, 인터럽트가 걸릴 경우 인터럽트 모드의 레지스터 집합을 사용하게 된다. 운영체제는 Register Set0 를 사용하게 되며, 다른 task 들은 Register Set1 ~ Register 3 을 사용하게 된다.

하드웨어로 구현된 context manager 는 보조연산장치 형태로 연결되어 사용되게 된다. 이

context manager 는 task 를 관리하고 스케쥴링하는 역할을 담당하게 된다.

명령어 집합(instruction set)의 경우 load/store 를 제외한 명령어들은 기본적으로 3 operand 로 동작하도록 되어 있으며, 명령어 집합에 간단한 RISC 명령어뿐만 아니라, DSP 의 특징 또한 채용하였다. address register 의 값을 register 의 값이나 immediate 값을 이용해서 post increment 또는 pre increment 해서 사용하는 것이 가능하도록 하였으며 loop 의 수행이 편리하도록 decrement and branch 명령을 지원하기 때문에 간단한 DSP 프로그램들은 DSP processor 를 따로 사용하지 않고도 효율적으로 실행할 수 있다. 본 논문에서 제안한 여러 레지스터 집합을 사용하기 위해서는 현재 사용하는 레지스터 집합을 다른 것으로 변경하는 있는 명령이 필요하게 되므로 이러한 기능을 수행하는 명령들이 명령어 집합에 추가되었다.

2.2 Multiple Register Sets

지금까지 운영체제는 작업전환(Context switching)을 위해 필요한 작업들을 소프트웨어적으로 처리하였다. 작업 전환을 위해서는 여러 가지 일들이 필요한데, 그 중 주된 오버헤드는 context 를 저장하고 복구하는 과정에서 발생하게 된다. 이 과정에서 걸리는 오버헤드를 줄이기 위해 프로세서 내부에 여러 개의 레지스터 집합을 사용하는 방법을 제안한다.

여러 레지스터 집합을 사용하게 되면, 많은 작업을 번갈아 가면서 실행할 때 효율적이지만, 레지스터 집합의 수가 늘어날수록 프로세서의 크기가 커지는 단점이 있게 된다. 이번에 설계한 프로세서의 경우 그림 2 또는 그림 3 과 같이 5 개의 레지스터 집합을 가지고 있다. 5 개의 레지스터 집합 중 세 집합은 일반적인 목적으로 사용되는 레지스터 집합(normal register set)이며, 한 집합은 스케쥴링 작업을 하기 위한 스케쥴링 레지스터 집합(scheduling register set), 나머지 한 개의 집합은 인터럽트가 걸린 경우에 사용하게 되는 인터럽트 레지스터 집합 (interrupt register set)이다.

각각의 집합은 일반적인 용도로 사용되는 GPR(general purpose resister)와 현재 프로세서의 상태 및 PC 값 등을 기억하기 위한 SPR(special purpose register)로 구성되어 있다. 프로세서가 어느 한 순간에 접근할 수 있는 GPR 과 SPR 의 수는 각각 32 개씩 이다. 각각의 레지스터 집합이 서로 영향을 주지 않도록 독립적으로 설계할 수도 있으나, 그렇게 할 경우 레지스터 집합을 구현하기 위해 많은 면적이 필요하기 때문에 각각의 집합이 사용될 목적에 따라서 독립적인 레지스터의 개수를 제한하여 사용하고, 나머지 부분은 서로 공유해서 사용하도록 구현하였다. 이렇게 구현할 경우, 프로세서의 크기를 줄일 수 있는 장점 이외에도 여러 작업 간에 서로 값을 전달하거나 공유해야 할 값이 있는 경우 서로 공유하고 있는 레지스터에 그 값을 저장함으로써 쉽게 데이터를 주고받을 수 있다는 장점도 가지게 된다.

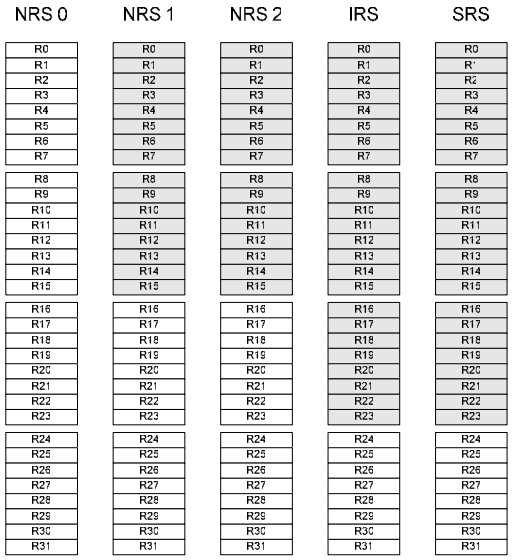

그림 2. General Purpose registers

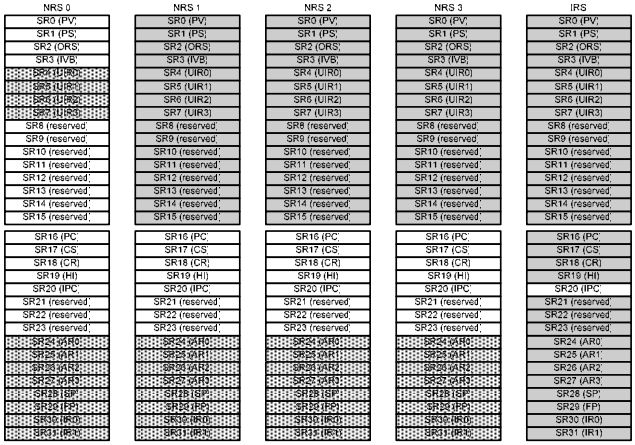

그림 3. Special Purpose registers

제안한 구조를 사용한 프로세서에서 NRS0 는 운영체제, NRS1 와 NRS2 는 자주 사용되는 task, IRS 는 interrupt, SRS 는 스케쥴링의 목적으로 사용되게 된다. 컴파일러를 이용해서 프로그램을 만드는 경우, 대부분의 경우에 8~16 개의 레지스터만을 사용하기 때문에 NRS 들은 GPR 중

R16~R31 의 16 개의 레지스터만을 독립적인 레지스터로 가지고 있게 되고, R0~R15 는 공유해서 사용하는 방식으로 구현하였다. IRS 와 SRS 의 경우에는 단순한 작업만을 수행하기 때문에 레지스터가 많이 필요하지 않게 되므로, GPR 중 R24~R31 에 해당하는 부분만을 독립적인 부분으로 가지고 있게 된다.

SPR 의 경우에는 SR0~SR15 를 모든 레지스터 집합이 공유해서 사용하게 되며, NRS 와 SRS 는 SR16~SR31 을 독립적으로 가지고 있다. 이 부분의 레지스터들은 PC(Program Counter), CS(Context Status), CR(Condition Register), EPC(Exception PC), multiplication 결과의 상위/하위 결과인 HI (multiplication High Result), LO(multiplication Low Result)의 값을 저장하는 목적으로 사용된다. IRS 의 경우에는 EPC 만을 독립적으로 가지고 있게 되는데 이러한 이유는 간단한 interrupt routine 만을 수행하고 인터럽트가 걸리기 전으로 돌아가게 되기 때문이다.

제안한 여러 레지스터 집합을 사용하는 구조에서 어떤 특정 레지스터 집합에서 다른 레지스터 집합으로 변경하는 경우는 다음과 같이 나누어 볼 수 있다.

<표 1> Register Set Change

From	To	condition
NRS0 ~ 2	IRS	by high priority IRQ
IRS	NRS0 ~ 2	By RETI (return from interrupt)
NRS0 ~ 2	SRS	By SCH or sche.int. (schedule instruction or schedule interrupt)
SRS	NRS0 ~ 2	By SWC (switch register set instruction)
IRS	SRS	Not permitted

IRS 에서 SRS 로의 레지스터 집합으로의 변환과 NRS 에서 NRS 로 바로 변환하는 것은 허용되지 않는다. 이러한 이유는 IRS 의 경우 인터럽트에 대한 간단한 루틴만을 실행하는 것이 원래 목적이며, 위에서 언급한 두 경우에 대해 레지스터 집합을 임의로 변경하는 것을 허용할 경우, 운영체제가 현재 어떤 컨텍스트가 실행되고 있는지를 알아야 할 필요가 있는데, 이를 유지하는 것이 어렵게 되기 때문이다.

제안한 여러 레지스터 집합 구조를 사용하지 않고 소프트웨어적으로 작업전환을 하게 되는 경우에는 기본적으로 사용하던 레지스터와 PC 값들을 메모리에 저장하고 복원시켜 주어야 한다. 따라서 최소한 33 번의 store / load 연산이 필요하게 된다. 이때 사용하는 메모리는 보통 한 칩에 프로세서와 함께 내장된 것이 아니라 외부에 연결된 것이기 때문에 메모리에 데이터를 쓰거나 메모리로부터 데이터를 읽기 위해서 여러 사이클이 필요하게 된다. 즉 한번의 작업 전환을 하기 위해서 필요한 오버헤드가 수백 사이클에 이르게 된다. 그러나 제안한 레지스터 집합 구조를 사용하는 경우에는 load / store 하는데 필요한 오버헤드가 없게 되며 단지 현재 사용하는 레지스터 집합을 다른 것으로 바꾸라는 명령어를 한번만 사용해주면 되게 되므로 1 사이클이 필요하게 된다. 즉 제안된 구조를 사용하면 작업전환(context switch)이 이루어질 때 소비되는 지연시간이 기존의 소프트웨어적인 방식에 비해 월등히 빨라지게 되므로, 실시간 시스템에 유리한 장점을 가지게 된다.

2.3 Hardware Context Manager

지금까지 운영체제에서 소프트웨어적으로 context manage 를 구현해 왔었다. 그러나 이 방식의 경우 스케쥴링을 하기 위해서 ready 상태에 있는 context 의 리스트를 관리해야 하며, 매번 리스트에 들어 있는 context 중에서 가장 먼저 실행해야 할 것을 찾아야 한다. 이러한 과정에서 메모리를 자주 사용하게 되기 때문에 많은 오버헤드가 걸리게 되는 단점이 있다. 범용 프로세서와 같이 많은 프로세스들이 동작해야 하는 환경에서는 위와 같이 구현하는 것이 효과적이겠지만, 임베디드 시스템에서는 동시에 실행하는 context 의 갯수가 범용 프로세서처럼 많은 것이 아니라 제한적이게 된다. 따라서 context manage 를 소프트웨어적으로 구현하지 않고 하드웨어로 구현하는 것이 가능하며, 이렇게 할 경우, 실시간 시스템에서는 작업 대상을 바꿀 때마다(context switch) 걸리던 지연 현상을 쉽게 해결할 수 있게 된다.

기존의 소프트웨어로 처리하는 방식과 하드웨어로 처리하는 방식을 비교해 보면 다음과 같다.

1) Scheduling

기존의 소프트웨어적인 방식에서는 실행해야 할 context 의 목록을 가지고 있으면서 매번 그 목록 중에서 가장 먼저 실행해야 할 context 를 선택하게 된다.

하드웨어로 구현하는 경우 각각의 context 에 대해서 Ready register 를 정하고, 각각의 context 에 대한 정보를 담고 있는

레지스터의 값을 간단한 논리 회로를 이용해서 비교하여 선택할 수 있다.

2) Delaying a context

기존의 소프트웨어적인 방식에서는 적은 수의 timer 를 사용하거나, 운영체제가 매 단위 시간(tick)마다 숫자를 감소시키면서 지연시간을 관리하게 된다.

하드웨어로 구현할 경우에는 각각의 context 마다 timer 또는 counter 를 두고 사용할 수 있다.

3) Event

소프트웨어적인 방식에서는 역시 event 를 목록으로 관리하면서 event 가 발생하면 해당하는 목록을 탐색하면서 context 들을 wake 하게 된다.

하드웨어로 구현하는 경우에는 각각의 context 마다 레지스터를 두고 event 가 발생하면 해당하는 레지스터의 값을 변경함으로써 wake 할 수 있다.

설계한 프로세서에서는 제안한 Hardware Context Manager 를 보조프로세서로 인식하고 사용할 수 있도록 하였다. 보조프로세서 형태로 구성함으로써 운영체제가 지원하는 경우에만 사용하고 지원하지 않는 경우에는 소프트웨어적으로 처리하는 등의 변경이 쉬워지게 된다.

제안하는 Hardware Context Manager 의 전체 구성은 그림 4 와 같다.

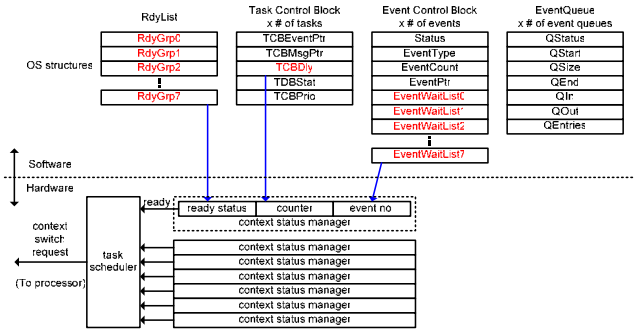

그림 4. hardware context manager

그림 4 의 상위 부분은 context 를 관리하는 자료 구조이다. Task 중에 ready 상태인 것을 가지고 있는 ready list 가 있고, 각 task 마다 task 의 상태를 나타내는 data 구조를 가지고 있어서, task 가 얼마나 지연된 후에 ready 상태가 될지 혹은 어떤 event 가 발생했을 때 ready 가 될 지를 나타낸다. 그리고 각 event 마다, 그 event 가 발생했을 때 ready 상태가 될 task 의 list 를 가지고 있다. 그림 4 의 하위 부분은 소프트웨어로 구현되었던 데이터 구조를 하드웨어로 구성한 것이다. task 마다 context status manager 가 있어서 각 status 의 상태를 관할한다. 소프트웨어에서 ready list 는 각 task 마다 ready 상태인지를 가리키는 레지스터를 둠으로써 대체한다. 소프트웨어에서는 각 task 마다 delay 값을 가지고 있고 timer 의 기준 시간(tick)마다 값을 줄였었는데, hardware context manager 에서는 context status manager 마다 counter 가 있어서 그 관리를 용이하게 하였다. 그리고 event list 를 대신해서, 각 context status manager 에 event number 를 저장하는 레지스터를 두었다. 각 context status manager 는 counter 와 event number 를 보아서 task 의 상태를 결정한다. 각 task 의 상태를 이용해서 task scheduler 가 가장 우선순위가 높은 task 를 선택한 뒤 프로세서에 전달하게 된다.

Context status manager 는 그림 5 와 같이 구성되어 있으며 각 task 마다 할당되어 있다. Task 가 일정 시간 동안 지연될 때 사용하는 counter, event 를 기다리고 있을 때 사용하는 event number register, 그리고 외부에서 오는 신호를 기다릴 때 사용하는 external event logic 이다. 그리고 이 세 모듈의 결과를 종합해서 task 의 상태를 판별한다.

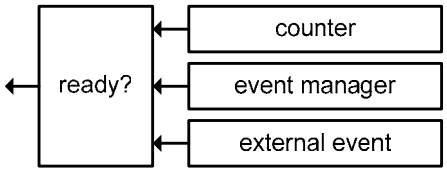

그림 5 Context status manager

제안한 구조를 사용하지 않고 소프트웨어적으로 context management 를 하는 경우에는 ready list 를 관리하기 위해 작업변경이 일어날 때마다 매번 메모리에 저장되어 있는 list 에 context 를 추가하거나 삭제해야 하고, 각각의 context 에 대해서 우선순위를 결정하기 위해 counter 값들을 변경시켜주는 작업 등을 수행해 주어야 한다. 이러한 과정은 모두 메모리와 연관된 것들이며, context 의 수가 많아질수록 관리하는데 더 많은 사이클이 소모되게 된다. 반면에 제안한 구조를 사용해서 context manager 를 하드웨어적으로 구현한다면 보조연산장치와 데이터를 주고받는데 필요한 오버헤드만이 존재하게 되기 때문에 소프트웨어적으로 관리하던 것에 비해 훨씬

빠르게 되므로, 지연 시간이 중요한 요소로 작용하는 실시간 임베디드 시스템에 적합한 구조라고 볼 수 있다.

제안한 구조를 이용해서 context manager 4 개를 포함하고, 64 개의 task 를 처리할 수 있는 hardware context manager 를 설계하여 검증에 사용하였다.

3 Implementation and synthesis result

Verilog HDL 을 통해서 RTL 로 기술한 후 매그나칩 0.25 μm CMOS 공정을 사용해서 Synopsis 의 Design Compiler 를 이용하여 합성하였다. 합성한 결과를 살펴보면 프로세서의 경우 Critical path 가 4.21ns 로 최대 230MHz 로 동작하였으며, equivalent gate 수는 123,000 gates 이다.

Hardware Context Manager 의 경우 critical path delay 가 3.91ns 로 최대 255MHz 로 동작하였으며, equivalent gates 수는 3800gates 이다.

<표 1> 합성 결과

	프로세서	Hardware Context Manager
최대 동작주파수	230MHz	255MHz
# of Equivalent gates	123,000	3,800

4 Conclusions

본 논문은 실시간 시스템에 효율적인 멀티 쓰레드 기반의 임베디드 프로세서의 구조를 제안하였다. 여러 개의 레지스터 집합을 사용하는 구조를 제안함으로써 Context switch 의 오버헤드를 줄이고, 보조연산장치 형태의 hardware context manager 를 제안함으로써 RTOS 에서 context scheduling 으로 인한 지연 시간을 줄일 수 있는 방법을 제시하였다. 제안된 구조를 사용하면 적은 지연시간을 필요로 하는 실시간 시스템에서 더 낮은 동작 클럭으로 기존의 임베디드 프로세서를 사용했을 때와 같은 성능을 낼 수 있기 때문에 프로세서 부가적으로 저전력의 효과도 볼 수 있다.

Acknowledgment

본 연구는 과학기술부와 산업자원부에서 지원하는 System IC 2010 사업의 과제로 수행하였다.

References

[1] Young-Don Bae, Seong-Il Park, and In-Cheol Park, “A Single-Chip Programmable Platform Based on a Multi-threaded Processor and Configurable Logic Clusters”, *IEEE Int. Journal of Solid-State Circuits*, vol.38, No, 10, Oct., 2003

[2] P. R. Nuth and W. J. Dally, “A mechanism for efficient context switching”, *in Proc. ICCD*, Sept. 1001, pp.301-304

[3] S. Storino, A. Aipperspach, J. Borkenhagen, R. Eickemeyer, S. Kunkel, S. Levenstein, and G. Uhlmann, “A commercial multithreaded RISC processor”, *in IEEE Int. Solid-State Circuits Conf. Dig. Tech. Papers*, Feb. 1998, pp. 234-235

[4] (2002)Intel IXP2850 Network Processor Product Brief. Intel Corp.[Online]. Avaliable: ftp://download.intel.com/design/network/ProdBrf/25213601.pdf

[5] Paul Kohout, Brinda Ganesh and Bruce Jacob, “Hardware Support for Real-time Operating Systems”, *in First IEEE/ACM/IFIP International Conference*, Oct. 2003, pp.45 - 51

[6] Eickemeyer, R., et al, “Evaluation of Multithreaded Processors and Thread-Switch Policies,” *International Symposium on High-Performance Computing*, Japan, Nov., 1997

[7] Adomat, I. Furunas, L. Lindh, and 1. Stamer. “Real-Time Kemel in Hardware RTU: A step towards deterministic and high performance real-time systems.” *In Proceedings of Eighth Euromicm Workshop on Real-Time Systems*, L’Aquila, Italy, June 1996,pp. 164-168

[8] V. J. Mooney and D. M. Blough, “A Hardware-Software Real-Time Operating System Framework for SOCs,” *IEEE Design and Test of Computers*, pp. 44-51, November-December 2002.

[9] V. Mooney and G. De Micheli, “Hardware/Software Codesign of Run-Time Schedulers for Real-Time Systems,” *Design Automation of Embedded Systems*, pp. 89-144, September 2000