論文95-32B-5-5

파이프라인 RISC 프로세서에서 분기지연을 감소시키는 하드웨어 구조

(A Hardware Scheme to Reduce the Branch Penalty in Pipelined RISC Processors)

趙 鐘 顯 , 趙 榮 一

(Jong Hyun Cho , and Young Il Cho)

요 약

스칼라 RISC(Reduced Instruction Set Computer)프로세서에서 분기 명령어에의한 제어종속은 분기의 결과가 결정되어 타겟 주소가 반입될때까지 파이프라인을 지연 시키는 요인이된다. 본 논문은 이러한 제어 종속을 해결하기위해 모든 명령어의 반입 단계를 중복 시키는 하드웨어를 제안한다. 파이프라인 프로세서에서 인터럽트 문제는 한 명령이 종료되기전에 다음 명령들이 진행중임으로인해 인터럽트 발생시 올바른 프로세서 상태를 보존하는데 어려움이 따른다. 제안한 구조는 precise 인터럽트를 지원하며 프로세서 상태들을 보존하기 위해 필요한 버퍼를 갖는다. 성능 측정을 위하여 2개의 프로그램 코드를 사용하여 지연 분기(Delayed Branch)의 경우와 비교했으며,그 결과 분기지연을 획기적으로 줄일 수 있었음을 알 수 있었다.

Abstract

Conditional branch instructions are a major obstacle to the increasing of RISC processor performance,because they can break the smooth flow of instructions: the issuing of instructions after a branch instruction must often wait until the condition is resolved. This paper proposes a hardware scheme which has a duplicated fetching logic to reduce the penalty imposed by conditional branch instructions. The proposed shceme has a buffer to maintain states of processor, which supports the precise interrupt. We make use of two code segments to test the performance and their results were compared with those of the delayed branch. We got the result that the proposed scheme reduces the branch penalty extremely.

I. 서 론

최적화된 명령어를 순차순서로 처리하는 스칼라 RISC 프로세서의 성능은 데이타 종속과 분기 명령어에 의한 제어 종속에 의해 크게 영향을 받는다. 데이타 종속은 실행 중인 명령어가 아직 완료되지않은 그 이전 명령어의 결과를 사용하고자 할때 발생하며 많은 문헌이 이를 다루어 왔다.[1] [2] [3] [14] 그러나 이보다 더 심각한 성능감소 요인은 프로그램내에 평균적으로 3-4개의 명령어 마다 존재하는 분기문에 의한 제어 종속이다. 이러한 제어 종속을 해결하는 방안으로는 소프트웨어적인 방법(Static Scheduling)과 하드웨어적인 방법(Dynamic Scheduling), 그리고 이 두가지를 함께 사용하는 방법등이 있다.[4] [5] [7] [8] [15] [16]

소프트웨어적인 방법은 분기 명령어 이후의 지연 슬롯(Delay Slot)에 분기와는 무관한 명령어를 채워서 분기 지연(Branch Delay)을 해결하는 지연 분기가 대표적이나, 지연 슬롯에 효과적으로 명령어를 채울 확

* 正會員, 水原大學校 電子計算學科
(Dept. of Computer Science , Suwon Univ.)

※ 이 논문은 1994년도 경기도 산학연 지역컨소시엄 연구비 지원에 의해 연구되었음.

接受日字:1994年4月25日 수정완료일:1995년4월27일

률이 극히 낮다는 단점이 있다. 한편 하드웨어적인 방법은 분기이후에 수행될 명령어를 예상하여 분기의 타겟 주소를 빠르게 반입하여 실행하는 브렌치 타겟 버퍼(Branch Target Buffer)가 있다. 그러나 분기예상이 부정확 할 경우 진행중인 명령어를 플러쉬(flush)시켜야 하며 새로운 타겟 명령어를 반입하는데 따르는 추가적인 부담을 동반 하게된다.보통 브렌치 타겟 버퍼의 경우 분기 예측 확률의 정확성은 85-90% 정도이다.[1][4][10][11]

파이프 라인화된 스칼라 RISC 프로세서는 한 명령의 실행이 종료되기 전에 다음 명령들이 반입되어 실행되므로 인터럽트가 발생 했을 경우,올바른 프로세서의 상태를 보존 해야하는 문제가 발생하는데 인터럽트 처리후 복귀 될때의 프로세서 상태들이 인터럽트를 발생시킨 명령어의 프로세서 상태들과 대응될때 이를 precise 상태라 하며[12], RISC 프로세서는 이러한 precise 인터럽트를 반드시 지원해야 한다. 본 논문에서는 분기 명령어에 의한 분기 지연을 해결하기 위하여 모든 명령어의 반입 단계를 중복 시키며,분기 명령어의 경우 분기의 두 경로를 동시에 반입하고 그중 한쪽 경로만을 선택하여 실행 시키는 하드웨어 구조를 제안한다. 또한 제안한 하드웨어는 precise 인터럽트를 지원하기위해 History 버퍼를 사용하여 인터럽트 이전의 프로세서 상태를 저장함으로써 precise 인터럽트를 지원한다.

본 논문의 성능 측정을 위하여, 제안한 하드웨어 구조와 지연 분기의 경우를 비교 분석하였으며, 분석결과 제안한 하드웨어는 분기 지연을 획기적으로 줄일 수 있었으며 반입 단계를 중복 시키는데 따르는 하드웨어의 추가 부담은 성능 향상을 고려해볼때 큰 부담으로 작용 하지는 않는다.

Ⅱ. 브렌치 타겟 버퍼와 지연 분기

이 장에서는 분기 지연을 해결하기위해 사용하고있는 대표적인 방법인 브렌치 타겟버퍼와 지연 분기 방법에 대해 기술하며 동시에 제안한 하드웨어 구조에서 사용할 브렌치 타겟 버퍼에 대해서도 언급한다. 브렌치 타겟 버퍼와 지연 분기 방법은 실제로 많은 프로세서에서 주로 사용되고 있으며 효과적으로 분기지연을 극복하기위해 많은 개발이 이루어져왔다.

1. 브렌치 타겟 버퍼의 구조

보통 브렌치 타겟 버퍼는 4-way set associative와 LRU(Least Recently Used)배치 전략을 사용한다. 브렌치 타겟 버퍼는 프로세서내에 확정된 크기로 고정되는데 항목(Entry)수가 많을수록 많은 조건분기문의 정보를 저장할 수 있기때문에 적중률을 높일 수 있지만 하드웨어 부담이 증가한다. 브렌치 타겟 버퍼 항목들은 taken 분기들을 성공적으로 예상하는 범위에 대해서만 사용하며 일반적으로 브렌치 타겟 버퍼에는 4가지 정보가 유지될 수 있는데, 분기 타겟 주소로부터의 명령어 바이트들, 분기 태그, 분기 타겟 주소 그리고 예상 정보등이 이에 포함된다. 브렌치 타겟 버퍼를 구현하는 방법은 여러가지가 있는데 그중 이절에서는 분리된 브렌치 타겟 버퍼와 Brian K.Bray & M.J.Flynn[5][9][10] 이 제안한 명령어 캐시에 브렌치 타겟 버퍼를 포함시킨 구조에 대해 알아 본다. 그림 1은 이러한 2가지 브렌치 타겟 버퍼의 구조를 보여주고 있다. 분리된 브렌치 타겟 버퍼 체계에 있어서 항목들의 수는 매우 중요하며, 이것은 항목수가 많으면 많은 분기 명령에 대해 항목을 할당할 수 있고,적중률을 높일 수 있기 때문이다.또한 명령어 캐시에 브렌치 타겟 버퍼를 포함시킨 구조에서 항목의 수는 한개의 브렌치 타겟 버퍼 항목당 명령어수와 관련 되었고,명령어 캐시 크기에 의존한다. 브렌치 타겟 버퍼 설계시 과거에는 성능을 나타내기 위해 적중률(Hit ratio)에 초점을 두었지만 실제 성능 향상에 더 많은 영향을 주는 것은 명령어 반입 장치가 올바른 주소를 얼마나 잘 예상 하느냐하는 점이다.

본 논문에서 제안한 하드웨어는 브렌치 타겟 버퍼를 사용하지만 분기 예상 확률에 의존하지 않고 단순히 분기이후에 반입될 타겟 명령어의 주소를 간직하는 용도로써만 사용될뿐이다.

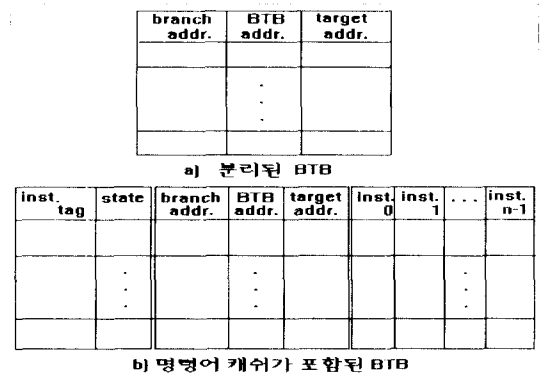

그림 1. 브렌치 타겟 버퍼를 구현하는 2가지 방법
Fig 1. Two methods to implement BTB.

2. 지연 분기

분기 이후의 지연 슬롯에 적절한 명령어를 스케줄함으로써 분기지연을 감소시키는 방법으로 그림 2는

지연 분기에 명령어가 스케쥴될 수 있는 가능한 3가지 방법을 보이고있다.[1]

지연슬롯에 삽입된 명령어를 지연 명령어(Delay Instruction)라 한다. 그러나 유용한 지연 명령어를 찾는 것이 항상 가능한 것은 아니어서 하나의 지연 슬롯에 명령어를 채울 확률은 약 80%에 이르지만 두번째 지연 슬롯을 채울 확률은 25%이하로 감소하게되어 보통 많은 지연 슬롯에 NOP 코드가 채워진다.[1][9] 또다른 문제점은 인터럽트가 발생했을 경우 지연 슬롯과 타겟명령의 프로그램 카운터(PC)값을 저장하기위해 여러개의 프로그램 카운터(지연 길이+1)가 필요한 점이다.[1]

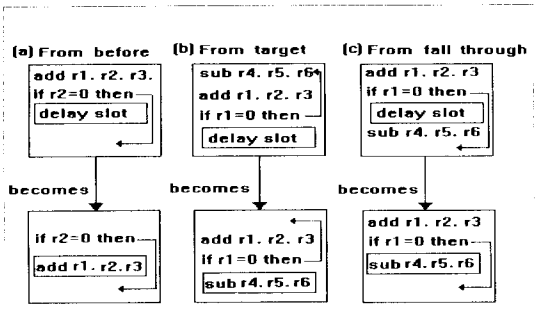

그림 2. 지연 슬롯의 스케쥴링
Fig. 2. Scheduling of branch-delay slots.

3. 제안한 브렌치 타겟 버퍼의 동작

본 논문에서 제안한 하드웨어는 분기의 taken 명령어를 위해 브렌치 타겟 버퍼구조를 사용하는데 기존의 브렌치 타겟 버퍼와는 다르게 분기 명령어의 주소와 taken 주소만을 항목으로 갖고 있다. 그림3은 본 논문에서 제안한 브렌치 타겟 버퍼의 구조이다.

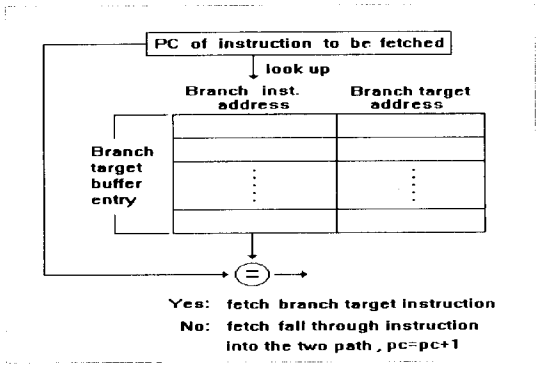

그림 3. 제안한 하드웨어의 브렌치 타겟 버퍼구조
Fig. 3. Structure of BTB in the proposed hardware scheme.

프로그램상의 명령어가 반입되어 해독될때, 비교 명령어를 만나게되면 분기에 따르는 양쪽경로의 명령어가 동시에 반입되는데 조건분기문 바로 다음에 있는 fall through 명령어는 프로그램 카운터의 증가 값을 사용하고 분기의 타겟 명령어는 브렌치 타겟 버퍼안의 대응되는 주소에 의존하게 되는 것이다.

Ⅲ. 파이프 라인

1. RISC 스칼라 프로세서의 파이프라인

어떤 프로그램을 파이프라인화 하는것은 단위시간당 완결되는 명령어 수(CPU Instruction Throughput)를 증가시키지만 명령어 개개의 수행시간은 파이프라인 제어에 필요한 요소로 인하여 약간 증가된다.즉 파이프라인된 프로그램은 수행의 고속화와 전체 수행시간의 단축을 의미하는 것이다.

RISC 스칼라 프로세서의 파이프 단계는 아래와같이 5 단계로 정의된다.[1][2]

1.IF -명령어 반입
2.ID -명령어 해독
3.EX -명령어 수행과 유효 어드레스 계산
4.MEM -메모리 엑세스
5.WB -WRITE BACK

그림4는 각 명령어 형태에 따른 수행 동작을 보여주고 있다. 매 파이프 단계는 매 클럭 사이클마다 활성화되며 이를 위해서는 한 파이프 단계의 모든 동작(Operation)은 1 클럭내에 완료되어야하고 어떠한 조합의 동작이라도 가능할 수 있어야 한다. 또 프로그램

Instruction	Clock cycle Number								
	1	2	3	4	5	6	7	8	9
Instruction i	IF	ID	EX	MEM	WB				
i+1		IF	ID	EX	MEM	WB			
i+2			IF	ID	EX	MEM	WB		
i+3				IF	ID	EX	MEM	WB	
i+4					IF	ID	EX	EM	WB

그림 4. 명령어의 파이프라인 단계
Fig 4. Pipeling stages of a instruction.

카운터는 매 클럭마다 반입 단계에서 증가되는데 이것은 별도의 증가기(Incrementer)가 있어야함을 말하며 프로세서 자원들 중 메모리 시스템에 관련된 자원이 파이프라인으로 인해 가장 큰 영향을 받는데 메모리 엑세스 시간이 변하지 않는다 하더라도 최대 메모리 반입폭(Band Width)은 파이프라인이 안된 프로세서

에 비해 5배로 증가되어야 한다.[1] 그 이유는 매 사이클마다 두개의 메모리 엑세스가 필요하기 때문이며 이를 위해 대부분의 프로세서들은 별도의 명령어 및 데이타 캐시를 사용한다.

2. 제안된 프로세서의 파이프라인

그림5는 제안한 하드웨어 구조에서의 파이프라인 단계를 보이고있다.이 구조의 기본 동작은 해독 단계 이전의 모든 파이프라인 단계를 중복시키는 것이다. 즉 5개의 파이프라인 단계를 갖는 모델의 경우 명령어 반입 단계를 중복시킨다는 것을 의미한다. 파이프라인에는 2개의 경로가 있으며 각각 경로1과 경로2로 지칭하며 일반명령어일 경우는 경로1(IF1)과 경로2(IF2)가 동일 명령어를 반입하게되며 비교-분기(Compare-Branch)명령어를 만나게되면 경로1은 fall through 명령어를 반입하고 경로2는 브렌치 타겟 버퍼의 타겟 명령어의 주소를 반입하게 된다. 분기의 결과는 경로1,2에 있는 서로 다른 명령어들이 해독 단계에 진입하기 전에 결정되며 결과 값에 따라 한쪽 경로의 명령어만이 실행되고 다른 쪽의 명령어는 단순히 버려지는 것이다.

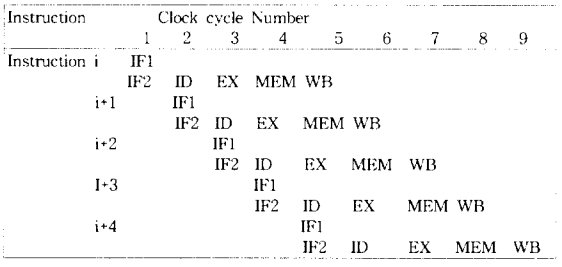

그림 5. 제안한 하드웨어 구조의 파이프라인 단계
Fig. 5. Pipeling stages in the proposed hardware scheme.

Ⅳ. 브렌치 타겟 버퍼를 갖는 제안된 프로세서의 구조와 동작

1. 비교-분기 명령어

RISC 프로세서에 있어서 보통 비교 명령어와 분기 명령어는 서로 독립적이 아닌 항목으로 취급된다.[13] 즉 조건 분기 명령어는 비교 명령어에 종속적이라는 것을 가정한다. 이것은 또한 파이프라인 단계가 다음에 수행될 명령어를 예상할 수 있는 능력을 갖게됨을 의미한다.

제안한 하드웨어는 명령어의 반입 단계를 중복시키고 분기명령어의 경우 한쪽 경로만을 선택해야 하므로 이를 위해 여분의 비트 CBB(Conditional Branch Bit)와 APB(Active Path Bit)비트가 필요하다. CBB 비트는 비교 명령어를 접했을때 이를 지시하기 위한 플래그로써 동작하고,APB 비트는 두 경로중 어떤 경로가 해독 단계로 진입할 것 인가를 나타내는 비트로써 사용된다. 프로세서는 디폴트로 경로 1을 수행하게되고 비교 명령어를 만날때마다 CBB비트를 세트시키며, 비교 명령어의 실행결과에 따라 APB 비트를 세트시킨다. 또 본 논문의 하드웨어 구조를 정상적으로 동작시키기위해 2상(Two-Phase)시스템 클럭이 필요하며 이것은 경로 1,2가 서로 독립적으로 동작하게한다. 즉 경로 1은 위상(Phase) 1에서 동작하며 경로 2는 위상 2에서 동작하게하여 1사이클에서 2번의 반입이 이루어진다. 그러나 이러한 동작을 위하여 추가적인 사이클을 필요로하지 않는다.

2. 프로세서의 명령어 처리

프로세서가 비교-분기 명령어를 접했을 때의 단계를 설명하면 다음과 같다.즉 프로세서는 반입한 명령어가 비교 명령어일 경우 해독 단계의 끝에서 즉각적으로 CBB 비트를 세트시키게되며 이때 분기 명령어는 이미 반입되어있는 상태가 된다. 이후 비교 명령어의 실행 단계가 시작됨과 동시에 분기의 양쪽 경로에 존재하는 명령어가 반입되는데 한쪽은 분기의 fall through 명령어이며 다른 쪽은 브렌치 타겟 버퍼에 있는 타겟 명령어가 된다. 두 경로중 한쪽이 결정 되는 시점은 비교 명령어의 실행 단계 끝에서 APB 비트의 세트 여부에 따라 알 수 있으며 이상의 단계를 거치면 파이프 라인은 분기지연없이 명령어가 실행될 수 있는 것이다.

단 이경우는 조건분기문이 브렌치 타겟 버퍼안에 존재한다고 가정했을 때의 상황이며 일반적으로 브렌치 타겟 버퍼안에 해당명령이 존재할때와 존재하지않을 때를 고려하면 다음과 같다.

1) 조건 분기문이 브렌치 타겟 버퍼항목에 있을 경우 위에서 살펴본 바와 같이 프로세서는 파이프라인의 어떠한 지연없이 분기의 양쪽 경로를 반입하고 해독하게되며 두 경로중 하나가 선택된다.
2) 조건 분기문이 브렌치 타겟 버퍼항목에 없을 경우 프로세서는 CBB비트의 상태에 의해 브렌치 타겟 버퍼 항목을 참조하게 되는데 해당 분기문이 존재하지않을 경우 경로1이 선택되어 해독 단계로 진입한다. 만일 분기의 결과가 fall through 명령어를 지향했다면 이경우 역시 지연없이 처리될 수 있다. 그러나 분기의 결과가 타겟을 향한 것으로 판명되면 반입된 명령어를 즉

각적으로 취소 시키고 타겟 명령어의 유효 어드레스를 계산해야하며, 이것에 따르는 1사이클의 지연이 발생한다. 그리고 새로운 분기문은 브렌치 타겟 버퍼에 등록된다.

이상의 내용을 요약한 것을 그림 6에 나타내었다.

```
PROCEDURE Find_Entry
begin
  If (exist corresponding entry)
  then
    begin
      fetch target_inst. found in BTB into IF2
        and fall_through_inst. into IF1
      if ( APB) then decode IF2
      else  decode IF1
    end;
  else
    begin
      fetch fall_through_inst. into IF1 and IF2
      If ( APB) then
        begin
          kill fetched inst.
          fetch target_inst.
          update BTB
        end;
          decode IF1
    end;
end;
  reset APB, CBB
```

그림 6. 분기문 후 다음 명령어의 선택과정
Fig. 6. Selection process of next instruction following branch instruction.

V. 인터럽트 처리

이 장에서는 RISC 프로세서에서 발생하는 인터럽트 문제를 다룬다. 명령어들이 파이프라인 되었을때 인터럽트를 처리하는 문제는 소프트웨어적으로 해결 할 수 도있지만 이것은 복잡한 인터럽트 처리 소프트 웨어를 필요로한다. 보다 간단한 방법으로는 하드웨어로 하여금 인터럽트를 처리하게하는 것이다.

1. Precise 인터럽트

일련의 명령어에서 인터럽트가 발생하면 하드웨어나 소프트웨어,혹은 이 두가지의 조합으로 인터럽트된 프로세서 상태를 저장해야한다. 이때 프로세서의 상태란 보통 프로그램 카운터, 레지스터와 메모리 값 등으로 이루어 진다. RISC 스칼라 프로세서는 반드시 precise 인터럽트를 지원 해야하는데 precise 인터럽트는 아래의 요건을 만족해야한다.[12]

1) 저장된 프로그램 카운터 이전의 모든 명령어는 실행이 종료되고 프로세스의 상태를 올바르게 변경했어야 한다.
2) 저장된 프로그램 카운터 이후의 모든 명령어는 실행되어서는 안되며 프로세서의 상태를 변경해서는 않된다.
3) 인터럽트가 프로그램내의 명령어에의해 야기된 예외 조건(Exception Condition)일 경우 프로그램 카운터는 인터럽트된 명령어를 가리키고 있어야하며, 인터럽트된 명령어는 실행이 되었을 수도 있으며 안되었을 수도 있다.

위의 세가지 조건을 만족하지 않았을 경우 이를 imprecise 인터럽트라 한다.

2. 제안한 구조의 인터럽트 처리

그림 7은 파이프라인에서 인터럽트 발생가능한 명령어 시퀀스의 예이다.아래의 예에서 여러가지 경우의 인터럽트가 발생할수 있는데 LW의 데이타 페이지 부재(Data Page Fault,MEM에서 발생), ADD의 산술 인터럽트(EX에서 발생)와 ADD의 명령어 페이지 부재(Instruction Page Fault)가 발생할수 있다. 만일

Instruction	Clock cycle Number								
	1	2	3	4	5	6	7	8	9
Instruction i-3	IF1								
	IF2	ID	EX	MEM	WB				
i-2		IF1							
		IF2	ID	EX	MEM	WB			
i-1			IF1						
			IF2	ID	EX	MEM	WB		
i(LW)				IF1					
				IF2	ID	EX	MEM	WB	
i+1(ADD)					IF1				
					IF2	ID	EX	MEM	WB
i+2						IF1			
						IF2	ID	EX	MEM

그림 7. 인터럽트 발생 가능한 명령어 시퀀스
Fig. 7. Instruction sequence which may be occured interrupts.

어떤 명령어의 반입 단계에서 인터럽트가 발생했을 경우 인터럽트로부터 프로세서의 상태가 반환 될때 해당 명령어로 반환되는 것이 아니라 파이프라인의 마지막 WB단계에 있었던 명령으로 반환하게함으로써 precise 인터럽트를 처리한다. 따라서 제안한 구조는 인터럽트 이전의 상태를 저장하고 비교-분기 명령어의 사건을 기록하기 위한 History를 유지하는 것이 필요하다. 이것을 위해 버퍼를 사용하는데 이 버퍼는 경로1

과 경로2의 프로그램 카운터,CBB, APB를 포함하며 버퍼의 크기는 하나의 비교-분기 명령어를 완료하는데 필요한 사이클의 수에 의존한다. 즉 5개의 파이프라인 단계에서 버퍼의 크기는 4 가된다.

VI. 적용예 및 검토

지금까지 분기문에의한 분기지연을 최소화하기 위하여 분기의 양쪽 경로를 동시에 반입하는 하드웨어에 대해 알아보았다.이제 제안된 하드웨어 구조와 지연 분기 구조의 성능을 비교하기위하여 Trace driven 시뮬레이터를 사용하여 2개의 프로그램 코드를 적용해 본다. Trace driven 시뮬레이션은 프로세서 레지스터의 내용을 추적하지않으며 함수 유닛은 단지 명령어가 사용하는 오퍼랜드 값을 지연시키는 요소로 모델링하는 방법이다. 시뮬레이터는 5개의 파이프 단계를 갖는 가상의 RISC프로세서를 기본으로 하였고,각 파이프 단계는 한 사이클동안 수행되고,캐시미스는 발생하지않으며 모든 조건분기명령의 타겟주소는 이미 계산되어있는 것으로 가정하였다. 그림 8은 배열 요소에서 최대값과 최소값을 찾는 프로그램의 루프에대한 기호 코드이다.

```
표식      기호 코드                          주석
****************** loop start ******************
Label_0 :load  r12 <- mem(r31, 4) ;load u
         load  r0  <- mem(r31,8)  ;load v
         sub   cr7 <- r12,r0      ;r12-r0=cr7
               cmplz cr7
         br    Label_2            ;u>v
         sub   cr6 <- r12,r30
         cmplz cr6
         br    Label_1
         lr    r30 <- r12         ;max=u
Label_1 :sub   cr7 <- r0,r28
         cmpgz cr7
         br    Label_4            ;v<min
         lr    r28 <- r0          ;min=v
         b     Label_4
Label_2 :sub   cr6 <- r0,r30
         cmpgz cr6
         br    Label_3            ;v>max
         lr    r30 <- r0
Label_3 :sub   cr7 <- r12,r28
         cmpgz cr7
         br    Label_4
         lr    r28 <- r12
Label_4 :add   r29 <- r29+2
         sub   cr4 <- r29,r27     ;loop 변수 감소
         cmplz cr4
         br    Label_0            ;loop end
**************** more instruction **************
```

그림 8. 최대, 최소값 프로그램의 기호코드
Fig. 8. Code segment of program the max_min.

지연 분기의 경우 분기문이후에 2개의 지연 슬롯이 발생하고 본 논문에서 제안한 하드웨어의 브렌치 타겟 버퍼는 프로그램의 최초 실행시에는 어떠한 항목도 없음을 가정하며, 분기경로를 잘못예측 했을경우 1 사이클의 지연이 발생한다. 아울러 메모리 페이지 부재와 같은 인터럽트 발생 가능성도 배재했다.

위 기호코드에서 1000개의 배열요소를 임의로 발생하여 실행시킬때 약 5000개의 지연 슬롯이 발생하며 제안된 하드웨어의 경우 최초의 프로그램 실행시 브렌치 타겟 버퍼 항목 등록때의 지연은 5 사이클이 발생한다.

그림 9는 위 기호코드를 1000회 실행했을때의 평균 지연 사이클을 지연 분기의 경우와 제안한 하드웨어에서 브렌치 타겟 버퍼크기에 따라 계산한 사이클을 나타낸다.

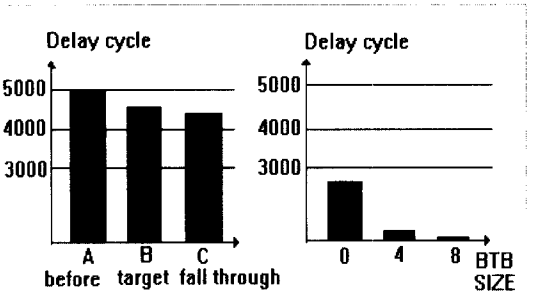

그림 9. 최대 최소값 프로그램에서 지연 분기와 제안한 하드웨어 구조의 평균 지연 비교
Fig. 9. Comparative results between delayed branch scheme and the proposed scheme for the max min program.

```
표식      기호코드                          주석
**************** loop start ******************
Label_0 :load   r6  <- mem(r31,4)   ;a[j]
         load   r7  <- mem(r31,8)   ;a[j+1]
         sub    cr7 <- r6,r7        ;cr7=r6-r7
         cmplz  cr7
         br     Label_1
         load   r8  <- r7           ;swap
         load   r7  <- r6
         load   r6  <- r8
         add    r1  <- r1+1         ;배열 index 증가
Label_1 :add    r0  <- r0+1         ;루프 index 증가
         sub    cr4 <- r12,r0
         cmpgt  cr4
         br     Label_0
************** more instruction **************
```

그림 10. 정렬 프로그램의 기호코드
Fig. 10. Code segment of sorting program.

그림 10은 또 다른 비교를 위해 사용한 정렬 프로그램의 루프부분이며 그림 11은 측정 결과를 보여준다.

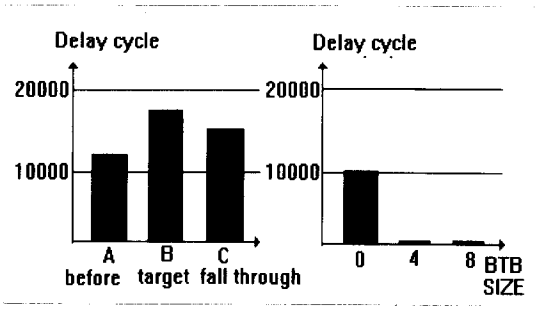

그림 11. 정렬 프로그램의 기호코드에 대한 평균 지연시간의 비교

Fig. 11. Comparative results between delayed brancd scheme and proposed scheme for the sorting program.

Ⅶ. 결 론

이상에서 살펴본 바와같이 지연분기의 경우 분기 경로가 taken일 경우 분기지연을 효과적으로 처리할 수 있는 대안이된다. 그러나 "If-else"구조의 경우 많은 분기경로가 발생 할 가능성으로 인하여 지연 슬롯이 증가하며 이경우 많은 지연슬롯이 NOP 코드로 채워진다.

본 논문에서 제안된 브렌치 타겟 버퍼를 갖는 하드웨어의 경우를 적용한 예에서 살펴 본 바와같이 브렌치 타겟 버퍼가 존재하지 않을 경우에도 지연분기보다 나은 성능을 유지할 수 있었으며 브렌치 타겟 버퍼크기가 증가할 수록 분기문에 의한 지연을 획기적으로 감소 시킬 수 있음을 알 수 있었다. 이것은 스칼라 RISC 프로세서의 최대 성능 향상이 1 머신 클럭에 1 명령어를 처리한다는 점을 감안하면 제안한 하드웨어는 최대 성능 향상에 접근 할 수 있는 가능성을 보여주고 있다.

또한 precise 인터럽트를 처리하기위해 진행중인 명령어들의 상태를 History버퍼를 사용하여 저장하며 크기는 필요한 상태 정보와 프로그램 카운터 값등을 수용하기위한 정도이므로 추가적인 하드웨어 비용이 큰 부담이되지 않는다.

앞으로의 과제로는 브렌치 타겟 버퍼의 타겟 명령어 적중률을 높이는 것과 동시에 2개 이상의 명령어를 반입하여 실행시키는 슈퍼 스칼라 프로세서[6][14]에 제안한 하드웨어 구조를 적용 시키는 방법에 대한 연구이다.

참 고 문 헌

[1] D. A. Patterson and J. L. Hennessy, Computer Architecture A Quantitative Approach, Morgan Kaufmann Publ., San Mateo, 1990.

[2] J. Ellis, "Bulldog : A compiler for VLIW architecture, Ph.D. thesis,Yale Univ. 1985.

[3] A. V. Aho, R. Sethi and J. D. Ullman, Compilers : Principles, Techniques, and Tools, Addison-Wesley, 1986.

[4] Gross T. R. and J. L. Hennessy, "Optimizing Delayed Branchs", Proceedings of IEEE Micro-15, pp 114-120, Oct., 1982.

[5] C. H. Perleberg and A. J. Smith, "Branch Target Buffer Design and Optimization", IEEE Trans. on Computers, Vol. 42, No. 4, pp 396-412, Apr. 1993.

[6] J. Fisher, "The VLIW machine : a multiprocessor for compiling scientific code", IEEE Computer, pp 45-53, Jul., 1984.

[7] R. Colwell et al., "A VLIW architecture for a trace scheduling compiler", IEEE Trans. computers, 37(8), pp 967-979, Aug., 1988.

[8] Mc Farling and John Hennessy, "Reducing the cost of branches", Proc. 14th symposium computer Architecture, pp 396-403, Jun. 1986.

[9] Brain. K. Bray and M. J. Flynn, "Strategies for Branch Target Buffers", Proc. of Micro-24, pp 42-50, 1991.

[10] Pradeep K. Dubey and M. J. Flynn, "Branch Strategies : Modeling and Optimization", IEEE Trans. on computers, Vol. 40., No. 10, pp 1159-1167, Oct. 1991.

[11] Tse-yu Yeh and Yale N. Patt, "Two-Level Adaptive Training Branch Prediction", Proc. 24th symposium on Micro architecture, pp 51-61, 1991.

[12] J. E. Smith and A. R. Pleszkum, "Implementation of precise interrupts in pipelined Processors", Proceedings, 13th Annual symposium on Computer Architecture, pp 396-404, June. 1986.

[13] M. J. Knieser and C. A. Papachriston, "Y-pipe:A conditional Branching Scheme without pipeline Delay", Proc. of Micro-25, pp 125-128, 1992.
[14] W. M. Johnson, "Superscalar processor design", Ph. D. thesis, Stanford Univ. 1989.
[15] M. S. Lam, "Software Pipelining : An Effective Scheduling Technique for VLIW Machines". In Proceedings of SIGPLAN '88 conference on programming Language Design and Implementation. pp 318-328, Atlanta, Georgia, Jan. 1988.
[16] R. B. Jones and V. H. Allan, "Software pipelining : An Evaluation of Enhanced pipelining", pp 82-92, ACM 1991.

저 자 소 개

趙 鍾 顯(正會員) 第 30卷 B編 11號 參照
현재 (주) A.T.I. 정보통신에 근무

趙 榮 一(正會員) 第 30卷 B編 11號 參照
현재 수원대학교 자연과학대학 전자계산학과 부교수