論文98-35C-2-1

정적 포워딩에 의한 VLIW 프로세서의 데이터 Hazard 처리

(Static Forwarding: An Approach to Reduce Data Hazards in VLIW Processor)

朴 炯 俊 *, 金 利 燮 *

(Hyoung-Joon Park and Lee-Sup Kim)

요 약

명령어 사이의 종속성은 프로그램상의 병렬성을 떨어뜨려 VLIW 프로세서의 성능을 저해하는 원인으로 작용한다. 특히 RAW(Read After Write) hazard는 발생 빈도가 빈번하기 때문에 이를 방지하기 위해서 일반적으로 포워딩 방법을 사용하였다. 하지만 VLIW 프로세서에서 포워딩을 구현하려면 많은 하드웨어 자원과 면적을 소모해야만 한다. 본 논문에서는 이러한 하드웨어 오버헤드를 줄이는 정적 포워딩을 제안한다. 정적 포워딩은 컴파일러 단계에서 RAW hazard를 검사하고, 포워딩 경로의 제어를 명령어 수준에서 지원하는 방법이다. 또한 포워딩 경로에 의한 면적을 줄이기 위하여 레지스터 파일을 변형하는 방법을 제시한다. 여기서는 정적 포워딩의 동작을 설명하고 기존 방법과 비교하였으며, VLIW 프로세서 모델을 사용하여 정적 포워딩을 검증하였다.

Abstract

To achieve high performance in VLIW processors, they must exploit the parallelism on application programs. Data dependency makes it difficult to find the instruction-level parallelism. Among the three kinds of data dependency, true dependency causes RAW(Read After Write) hazards that occur most frequently in VLIW processors. Forwarding is a widely used technique to reduce the performance degradation caused by RAW hazards. However, forwarding requires too much area of the chip when it is applied to VLIW processors. In this paper, static forwarding is proposed to reduce the hardware cost of forwarding circuits. It needs an extended compiler to detect RAW hazards and control the proposed forwarding scheme via instruction. And it uses the modified register file to shrink the area of forwarding path. VLIW Processor Model is also designed to verify static forwarding. This paper describes the operation of static forwarding and the comparison with the conventional forwarding.

I. 서 론

고성능 프로세서를 위해서는 주어진 하드웨어 자원을 효율적으로 사용할 수 있는 아키텍처를 설계하는 것이 중요하다. 하드웨어 자원을 효율적으로 사용하기 위해서는 프로세서가 수행하는 명령어들 간의 병렬성(ILP; Instruction Level Parallelism)을 충분히 활용하여야 한다.

명령어 수준의 병렬성을 활용하여 프로세서의 성능을 향상시키는 방법은 정적 스케줄링(static scheduling)과 동적 스케줄링(dynamic scheduling)의 두 가지로 분류할 수 있다. VLIW(Very Long Instruction Word) 프로세서는 컴파일러와 같은 소프트웨어를 통해서 프로그램을 최적화하는 정적 스케줄링 방법

* 正會員, 韓國科學技術員 電子工學科
(Dept. of Electrical Engineering Korea Advanced Institute of Science and Technology)

接受日字:1997年7月3日, 수정완료일:1998年1月30日

을 사용한다. 따라서 마이크로 프로세서가 최적화의 부담을 안게 되는 동적 스케줄링에 비해서 하드웨어 오버헤드(overhead)를 줄일 수 있다.

컴파일러가 프로그램을 최적화하기 위해서는 명령어 수준의 병렬성이 충분히 있어야 하고, 프로그램의 수행 형태를 예측하기 쉬워야 한다. 최근 들어 많은 관심이 집중되고 있는 멀티미디어, 디지털 신호 처리, 과학 계산 등의 응용 분야는 이러한 조건을 갖추고 있으며 고성능을 요구하기 때문에 VLIW 프로세서를 적용하기에 적합하다. 그러나 이러한 응용 분야에서도 명령어 수준의 병렬성을 극대화하는데는 한계가 존재한다[1], [2]. 명령어 사이의 종속성(dependency)은 VLIW 프로세서와 같은 다중 처리(multiple issue) 프로세서에서 명령어의 수행에 영향을 주게 된다. 그 중에서도 true dependency는 발생 빈도가 매우 높을 뿐만 아니라 명령어의 수행 순서와 시기를 결정 짓기 때문에 병렬성을 크게 떨어뜨린다[3].

True dependency를 해결하기 위해 일반적으로 포워딩(forwarding)이 사용된다. 포워딩은 하드웨어를 사용하여 종속관계의 명령어를 찾아내고, 포워딩 경로(forwarding path)를 통해 연산 결과를 지연 없이 기능 블럭에 전달한다.

이로 인하여 종속관계로부터 발생할 수 있는 RAW (Read After Write) hazard를 방지하고, 명령어 수준의 병렬성을 유지할 수 있게 된다. 하지만 VLIW 프로세서와 같은 다중 처리 프로세서에서 포워딩을 구현하는 것은 매우 큰 하드웨어 오버헤드를 필요로 한다[4].

본 연구에서는 이러한 하드웨어 오버헤드를 줄일 수 있는 정적 포워딩 방법을 제안한다. 정적 포워딩은 RAW hazard를 발생시킬 수 있는 종속관계의 명령어들을 컴파일 단계에서 찾아내고, 포워딩 경로의 제어를 명령어 수준에서 지원함으로써 포워딩 회로에 의한 하드웨어를 최소화한다. 또한 포워딩 경로에 의한 면적을 줄이기 위하여 레지스터 파일을 변형하는 방법을 제시한다.

II장에서는 hazard로 인한 VLIW 프로세서의 성능 변화와 기존 포워딩의 구현으로 인한 하드웨어 오버헤드를 알아본다. III장에서는 제안된 정적 포워딩에 대해 설명하고, IV장에서 정적 포워딩을 VLIW 프로세서에 적용하여 기존의 방법과 비교한다. V장은 결론이다.

II. VLIW 프로세서의 데이터 Hazard

1. Hazard로 인한 성능 저하

매 사이클 하나의 명령어를 처리할 수 있는 P단 파이프라인 프로세서의 이상적인 성능 향상율(speedup factor), $S_{pipeline}$은 P이다[5]. 하지만 hazard로 인하여 프로세서의 수행이 지연된다면 이와 같은 이상적인 성능 향상을 기대할 수 없다. 명령어에 대한 hazard 발생 빈도를 (α라 하고, hazard 발생시 k사이클의 파이프라인 정지(stall)를 필요로 한다면 실제적인 성능 향상은 다음과 같다.

$$S_{pipeline} = \frac{1}{(1-\alpha)+\alpha\times(1+k)} \times P = \frac{1}{1+\alpha k} P \quad (1)$$

$$\left(\alpha = \frac{\text{Number of hazards occured}}{\text{Number of instructions executed}}\right)$$

VLIW 프로세서에서의 성능 향상율은 다소 복잡해진다. 복수의 명령어가 한 사이클에 수행되므로 인해서 hazard를 발생시키는 명령어의 위치에 따라 서로 다른 성능 변화를 가져오기 때문이다. 프로세서가 매 사이클 I개의 명령어를 수행할 수 있도록 명령어 컴팩션(instruction compaction)이 되었을 때 얻어지는 최대(최소 성능의 식은 다음과 같다.

$$\max\{S_{VLIW}\} = \frac{1}{1+\alpha k} IP \quad (2)$$

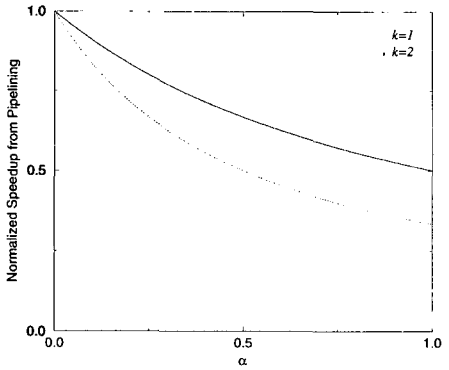
$$\min\{S_{VLIW}\} = \begin{cases} \frac{1}{1+\alpha Ik} IP, & 0 \le \alpha < \frac{1}{I} \\ \frac{1}{1+k} IP, & \frac{1}{I} \le \alpha \le 1 \end{cases} \quad (3)$$

그림 1은 α와 k에 대한 정규화된(normalized) 성능 향상율을 보여준다. 그림 1 (b)의 빗금 친 부분은 프로그램 특성에 따라 VLIW 프로세서에서 나타날 수 있는 성능 변화 범위를 나타낸다. 이것은 동일한 hazard 발생 빈도를 가지는 프로그램을 두 프로세서에서 수행시킬 때, hazard에 의한 성능 저하는 VLIW 프로세서에서 더 크게 나타날 수 있음을 의미한다.

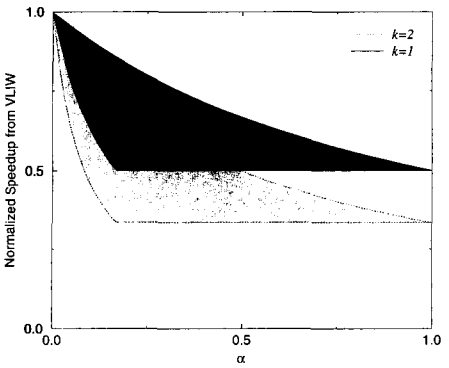
표 1은 실제 응용 프로그램에서의 성능 변화 예이다. 응용 프로그램은 VLIW 프로세서 모델에서[1] 수행되었으며, RAW hazard 발생시 1 사이클의 파이프라인 정지를 가정하였다. 나타난 결과에서 알 수 있듯이 RAW hazard는 매우 빈번하게 발생하며 이로 인한

[1] IV장 참조

성능은 이상적일 경우와 비교하여 50%~60% 밖에 되지 않는다.



(a) $S_{pipeline}$



(b) S_{VLIW}

그림 1. Hazard에 의한 성능 변화
Fig. 1. Speedup factor with hazards.

표 1. RAW hazard에 따른 VLIW 프로세서의 성능 향상율
Table 1. Speedup factor of VLIW processor depending on RAW hazards Application Program.

Application Program	α[2]	Simulated	Estimated[3]	
		S	max{S}	Min{S}
8×8 1D IDCT	0.42	12.0	16.9	12.0
16×16 ME	0.50	14.4	16.0	12.0

2. 포워딩 회로의 오버헤드

포워딩은 다음과 같은 두 단계를 거쳐서 이루어진다.

1. 레지스터 어드레스(register address)를 비교하여 명령어 사이의 RAW hazard 가능성을 검사한다.
2. RAW hazard가 발생하는 소스 오퍼랜드(source operand)에 대해서는 포워딩 경로를 통해서 결과 값을 전달한다.

첫 번째 단계를 위해서 레지스터 어드레스를 비교할 수 있는 비교기(comparator)가 필요하며, 두 번째 단계를 위해서 결과 값을 전달할 수 있는 포워딩 경로가 있어야 한다. N개의 기능 블럭을 가지는 프로세서에서 포워딩을 구현할 때 필요한 하드웨어는 다음과 같다[6].

Number of comparators
=Number of result tags×Number of source tags
×Number of pipeline stages between ID and WB
$=N\times 2N\times d$ (4)
$=2dN^2$

Forwarding path complexity
=Number of result buses×Number of source buses
×Number of pipeline stages between ID and WB
$=2dN^2$ (5)

여기서 d는 파이프라인에서 ID(Instruction Decode)단과 WB(Write Back)단 사이에 존재하는 파이프라인 단의 수이다.

식 4와 식 5를 바탕으로 포워딩 회로가 칩(chip) 면적에 미치는 영향을 유도할 수 있다. 기능 블럭 하나의 면적을 A라 하고 ε을 기능 블럭 면적에 대한 포워딩 회로 면적의 비라고 하면, 포워딩 회로의 면적은 εA이다. 포워딩 회로를 포함한 N개의 기능 블럭의 면적은 아래와 같이 나타낼 수 있다.

Estimated total area in N function blocks
=Forwarding circuits area+Function block area
$=N2\times \varepsilon A+N\times A$
$=\varepsilon A^2+AN$ (6)

그림 2는 COMPASS사의 0.6μm VTI library를 사용하여 기능 블럭과 포워딩 회로의 실제 면적을 나타낸 그래프이다. 막대 그래프로 그려진 부분이 datapath compile을 통해 얻어진 면적이며, 실선으로 나타난 부분은 식 6으로부터 예측된 결과이다. 실제 면적과 예측된 면적이 근사적으로 일치함을 알 수 있다. 막대 그래프에서 볼 수 있듯이 기능 블럭의 수가 증가함에 따라 포워딩 회로가 차지하는 면적이 N^2에 비례하여 증가하고 있다. VLIW 프로세서는 기능 블럭의 수가 단일 처리(single issue) 프로세서에 비해 많기 때문에 포워딩이 차지하는 면적의 비중이 커지게

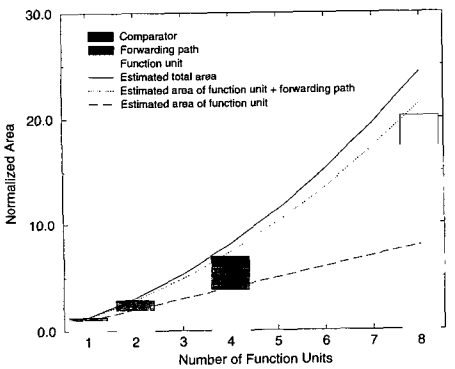
[2] due to RAW hazards
[3] estimated from equation 2 and 3

된다. 이는 곧 하드웨어 오버헤드의 증가를 의미한다.

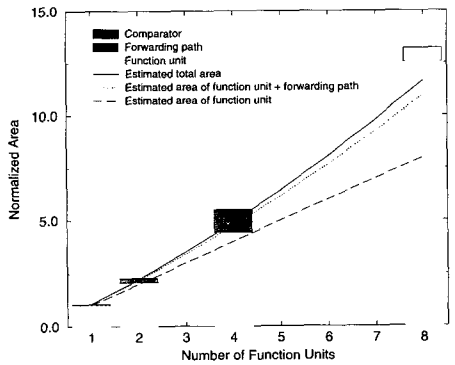
Ⅲ. 정적 포워딩

1. 정적 포워딩을 위한 컴파일러

정적 포워딩이란 프로그램이 수행되기 이전인 컴파일 단계에서 RAW hazard를 검사함으로써 하드웨어의 추가를 줄이는 방법이다. 그림 3은 기존 포워딩과 정적 포워딩과의 개념상 차이를 보여준다. 기존의 포워딩 회로는 비교기의 결과를 포워딩 경로 제어 신호로 사용하였다. 하지만 정적 포워딩에서는 컴파일러가 RAW hazard를 검사하고 그 결과에 따라 포워딩 경로 제어 신호를 명령어에 추가함으로써 포워딩을 구현한다. 따라서 그림 3의 점선으로 둘러 쌓인 부분을 절약할 수 있게 된다.



(a) ALU 블럭



(b) Multiplier 블럭

그림 2. 기능 블럭의 면적과 포워딩 회로의 면적
Fig. 2. Areas of function block and forwarding path.

컴파일러를 사용하여 RAW hazard를 검사하는 것은 하드웨어를 사용한 포워딩 방법과 동일한 알고리즘을 사용한다. 최적화 기능을 가진 대부분의 컴파일러는 데이터 종속성 및 hazard 검사가 가능하므로 여기서 더 이상 언급하지 않는다. 컴파일러가 RAW hazard를 발견한 후, 제어 신호를 생성하는 방법 역시 하드웨어와 동일한 알고리즘을 사용한다. 이러한 제어 신호를 명령어에 효율적으로 추가하기 위한 방법은 Ⅲ-2절에서 다룬다.

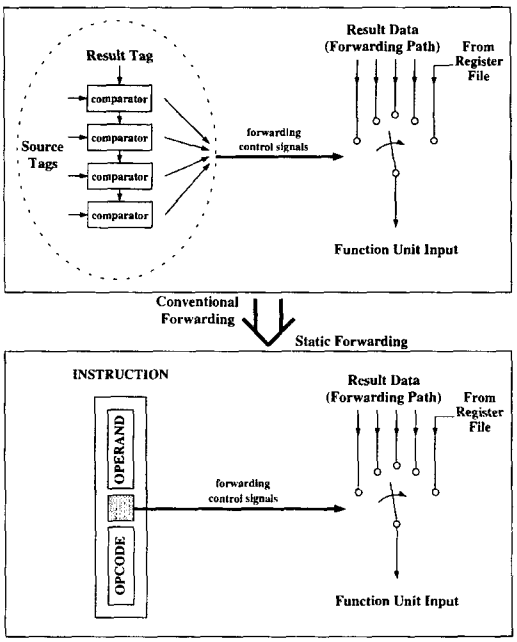


그림 3. 정적 포워딩의 개념도
Fig. 3. Conceptual diagram of static forwarding.

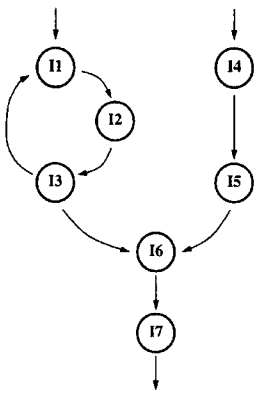
일반적으로 컴파일러는 프로그램의 수행 과정을 정확히 예측할 수 없다. 프로그램 수행 도중 조건 분기 명령어를 만나게 되면, 상황에 따라 그 다음에 수행하게 되는 명령어가 달라지게 된다. 또한 인터럽트(interrupt)나 예외(exception)가 발생하면 서비스 루틴으로의 분기가 일어나는 데, 이 경우는 인터럽트 등이 발생할 시점이 정해져 있지 않기 때문에 어느 명령어에서 분기가 일어날 지 예측할 수 없다. 따라서 컴파일러를 통해 정적 포워딩을 구현하기 위해서는 특별한 주의가 필요하다. 이와 같은 경우에 대해서 올바른 정적 포워딩을 수행하기 위한 방법은 다음과 같다.

(1) 분기 명령어에서의 정적 포워딩

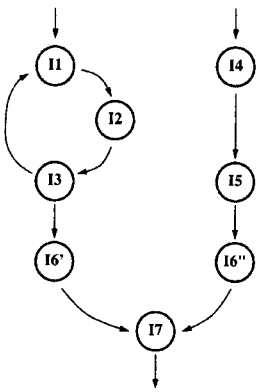
만일 어떤 명령어가 프로그램에서 두개 이상의 명령어로부터 수행된다면, 컴파일러는 복수 개의 명령어에 의해 발생될 수 있는 모든 RAW hazard를 해결할

수 있도록 포워딩 제어 신호를 생성해야 한다. 그림 4 (a)에서 I1과 I6 명령어는 최대 두개의 명령어와 종속 관계를 가질 수 있다. I1은 그 이전의 명령어와 I3, I6의 경우는 I3나 I5와 hazard를 일으킬 가능성이 있다.

하지만 명령어의 데이터 폭이 정해져 있으므로 모든 경우에 해당하는 포워딩 제어 신호를 포함할 수는 없다. 이를 위해서 종속 관계를 단순화하는 작업이 필요하다. Trace scheduling[7] 의 bookkeeping 방법을 활용하면 복수개의 종속 관계를 하나로 통일하는 것이 가능하다. 그림 4 (b)와 같이 I6를 I3와 I5 각각에 대하여 hazard를 해결할 수 있는 제어 신호를 포함하도록 I6′과 I6″으로 변형한 후, 분기가 일어나는 위치를 I3와 I5에서 I6′과 I6″으로 바꾸어 주면 된다. I6′과 I6″은 I7과 동일한 종속 관계를 이루므로 분기에 의한 복수의 RAW hazard가 단순화되었다.



(a) bookkeeping 이전의 명령어 실행 흐름



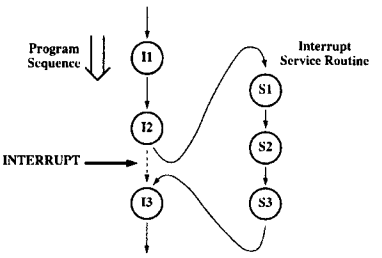
(b) bookkeeping 이후의 명령어 실행 흐름

그림 4. 명령어 실행 흐름도
Fig. 4. Flowgraph of instruction execution.

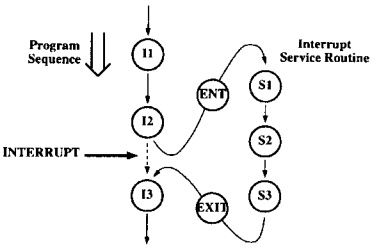
모든 분기 명령어에 의해 복수의 종속 관계가 발생하는 것은 아니므로 필요한 명령어에 대해서만 bookkeeping을 사용하면 된다.

(2) 인터럽트에서의 정적 포워딩

파이프라인 구조의 프로세서에서 인터럽트나 예외의 발생은 프로세서의 상태(status)를 변화시키므로 프로그램의 올바른 수행을 위해서는 상태 변화를 상쇄시켜 주어야 한다[8]. 정적 포워딩에서는 포워딩 제어 신호도 인터럽트에 의한 영향을 받게 된다. 그림 5 (a)는 프로그램 수행 중 인터럽트가 발생한 예이다. I3가 I2의 결과 값을 포워딩 경로를 통해 얻도록 제어 신호가 결정되어 있었다면, 인터럽트로 인해 서비스 루틴(service routine)을 수행한 후 I3는 S3의 결과를 포워딩 받게 된다.



(a) 인터럽트 발생과 처리의 예



(b) 정적 포워딩을 고려한 인터럽트 서비스 루틴

그림 5. 인터럽트 발생과 인터럽트 서비스
Fig. 5. Interrupt and interrupt service.

이와 같은 잘못된 수행을 방지하기 위해서 다음 같은 세 가지 방법을 사용할 수 있다.

1. 인터럽트가 발생한 시점에서 I2의 결과 값을 저장하였다가, 인터럽트 서비스 루틴을 마친 후 그 결과 값들을 포워딩 경로를 통해 다시 내보내 준다.
2. 인터럽트가 발생한 시점에서 I2의 결과가 저장될 레지스터 어드레스를 기억하였다가, I3가 수행될 때 I3의 포워딩 제어 신호를 무시하고 기억된 레지스터 어드레스로부터 얻어진 레지스터 값을 읽

는다.

3. 명령어를 사용하여 방법 1과 같은 기능을 수행한다.

방법 1, 2는 추가적인 하드웨어를 필요로 하지만 성능 면에서 우수하다. 방법 3은 명령어를 사용한다는 점에서 하드웨어의 추가는 필요 없으나 인터럽트 서비스 루틴의 명령어 추가로 인해 수행 시간이 길어진다. 그림 5 (b)는 방법 3에 의한 인터럽트 서비스 루틴을 보여준다. 서비스 루틴의 시작과 끝에 I2의 포워딩 값을 저장하였다가 내보내 주는 ENT, EXIT 명령어가 추가되었다. ENT는 기본적인 메모리 저장 명령어나 일반적인 연산 명령어 등을 사용하여 포워딩 값을 메모리나 레지스터에 저장한다. EXIT은 ENT에 의해 저장된 값을 읽어 포워딩 경로를 통해 내보내 주게 된다. 성능과 하드웨어를 고려할 때, 방법 2를 사용하는 것이 가장 효율적이다.

2. 정적 포워딩의 제어 신호

F개의 포워딩 경로를 가지는 프로세서에서 인코드(encode)된 제어 신호의 최소 bit 폭은 $\log_2 F$이다. 소스 오퍼랜드로 레지스터 값이나 포워딩 경로의 값을 읽도록 선택해 주는 신호를 포함한다면 $\log_2 F+1$이다. 일반적으로 한 명령어에 두개의 소스 오퍼랜드를 사용하므로 명령어 당 추가되어야 할 제어 신호는 $2(\log_2 F+1)$ bit이다.

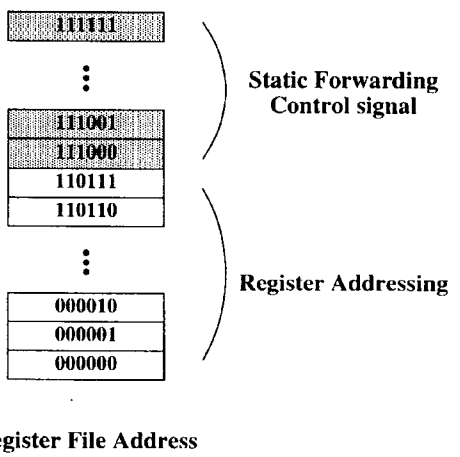


그림 6. 정적 포워딩을 위한 레지스터 어드레스의 할당
Fig. 6. Register allocation for static forwarding.

레지스터를 선택하기 위한 어드레스의 일부를 제어 신호에 필요한 bit으로 활용한다면 명령어에 별도의 bit을 추가하지 않고도 정적 포워딩을 위한 제어 신호를 첨가할 수 있다. 그림 6은 64개의 레지스터를 갖는 프로세서에서 8개의 포워딩 경로를 가질 경우 레지스터 어드레스 중 최상위 8개를 정적 포워딩을 위한 제어 신호로 할당한 그림이다.

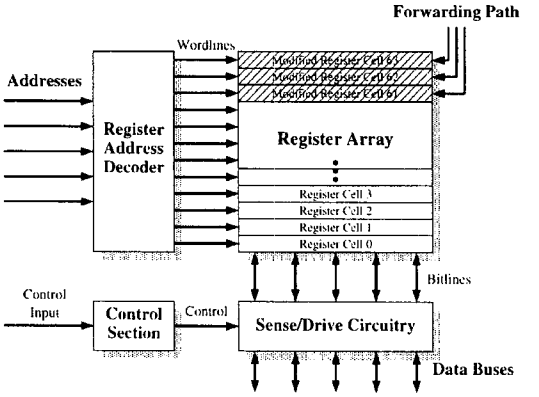
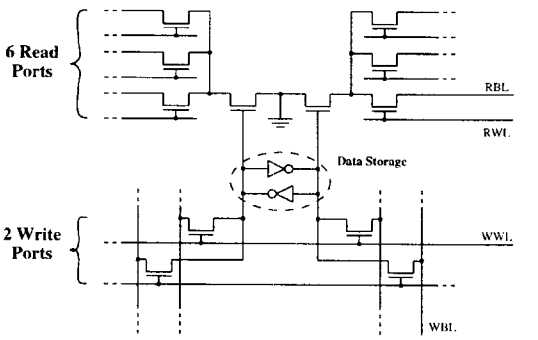
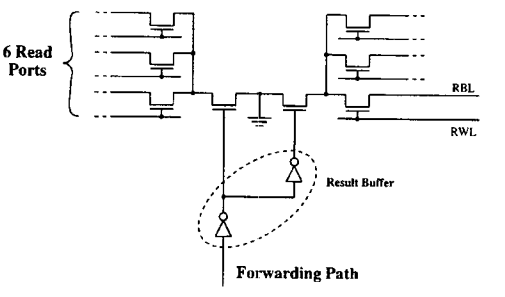


그림 7. 정적 포워딩을 위해 변형된 레지스터 파일
Fig. 7. Modified register file for static forwarding.



(a) 기본 레지스터 셀회로



(b) 변형된 레지스터 셀회로

그림 8. 정적 포워딩을 위해 변형된 레지스터 셀
Fig. 8. Modified register cell for static forwarding

이와 같은 형태로 정적 포워딩의 제어 신호를 할당할 때, 레지스터 파일의 일부를 변형함으로써 포워딩

경로를 선택해 주기 위한 다중화기(multiplexor)와 결과 값을 포워딩하는 포워딩 경로를 절약할 수 있다. 그림 7은 그림 6과 같은 레지스터 어드레스 구조를 바탕으로한 새로운 레지스터 파일을 보여준다. 그림 7에서 빗금 친 레지스터 셀들은 그림 8 (b)와 같다. 레지스터 셀을 일부 변형함으로써 데이터 저장기능을 없애고 포워딩을 위한 버퍼기능을 가지도록 하였다.

이와 같은 구조의 레지스터 파일을 사용하면 포워딩 경로가 레지스터 파일의 입출력 버스와 공유되므로, 포워딩 경로를 위한 면적을 줄일 수 있다. 또한 레지스터 파일의 어드레스 디코더를 사용하므로 각 기능 블럭에서 포워딩 경로를 선택하는 다중화기가 없이도 정적 포워딩을 구현할 수 있다.

Ⅳ. 성능 평가

본 연구에서 제안된 정적 포워딩 방법을 검증하고자 VLIW 구조를 기반으로 하는 VLIW 프로세서 모델(VLIW Processor Model, 이하 VPM)을 설계하였다. VPM의 구조는 다음과 같은 특징은 갖는다.

- 정적 포워딩
- 포워딩 회로를 절약하기 위한 변형된 레지스터 파일

56개의 24-port 범용 레지스터(16 read ports, 8 write ports)

8개의 register mapped 포워딩 경로

- 6 IPC(Issues Per Cycle)
- 4 단계 파이프라인
- 16개의 기능 블럭

4개의 ALU block
4개의 multiplier block
4개의 shifter block
1개의 vector sum block
2개의 load/store block
1개의 branch block

- 85개의 명령어 집합
- 조건 실행 명령어
- 32-bit 데이터 버스 폭

기존의 포워딩 방법과 비교를 위해 똑같은 사양을 가지면서 하드웨어 포워딩을 사용한 VPM도 같이 설계하였다. Verilog hardware description language를 사용하여 VPM을 합성(synthesis)이 가능한 behavioral model로 기술하였고, 정적 포워딩의 동작을 검증하기 위하여 응용 프로그램의 커널(kernel)을 실행시켰다. 기존의 포워딩 방법과 정적 포워딩 방법의 하드웨어 비용을 측정하기 위하여, VPM의 기능 블럭들을 각각의 방법에 따라 설계한 후 COMPASS tool을 사용하여 datapath를 compile하였다. 3-metal 0.6(m VTI library를 사용하였다.

1. 포워딩 방법에 따른 면적 비교

그림 9는 포워딩 방법에 따른 각 기능 블럭의 게이트 수를 나타낸다. ALU, multiplier, shifter 기능 블럭은 4개를 하나로 묶어 각각 M4ALU, M4MUL, M4SFT로 표기하였으며, 두개의 load/store 블럭은 각각 LDSTX, LDSTY 이다. COMP는 기존 포워딩의 비교기에 해당한다. 전체 게이트 수는 기존의 포워딩 방법이 정적 포워딩에 비해 42.2% 더 많이 필요한 것으로 나타났다. 이것은 기존 포워딩 방법이 상당한 하드웨어 오버헤드를 가지고 있음을 의미한다. 비교기의 게이트 수는 정적 포워딩 전체 기능 블럭 대비 6.1%로서 컴파일러가 RAW hazard를 검사함으로써 얻어지는 이득은 그다지 크지 않은 것으로 나타났다.

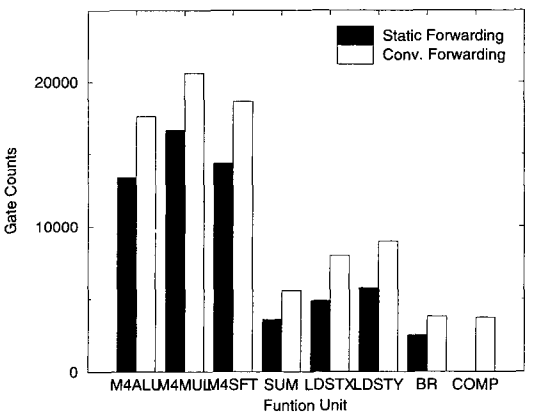


그림 9. 포워딩 방법에 따른 게이트 수
Fig. 9. Gate counts for two forwarding methods.

그림 10은 각 기능 블럭의 면적을 비교한 것이다. 면적에서 기존의 포워딩 방법은 정적 포워딩에 비해 60% 더 큰 면적을 가짐을 보였다. 비교기의 면적은 정적 포워딩 전체 기능 블럭 대비 5.3%으로, 비교기를 줄임으로써 얻는 효과에 비해 포워딩 경로를 레지스터 파일 버스와 공유함으로써 얻어 지는 이득이 훨씬 크다는 것을 알 수 있다. 특히 면적 비교가 게이트

비교보다 훨씬 큰 차이를 보인 것은 포워딩 경로의 배선에 필요한 면적이 매우 크다는 것을 의미한다.

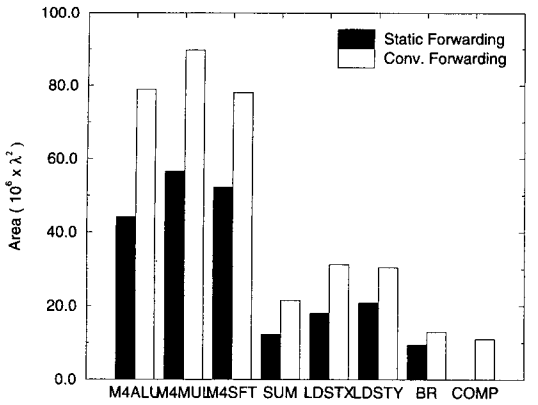


그림 10. 포워딩 방법에 따른 면적 비교
Fig. 10. Area comparison between two forwarding methods.

2. 포워딩 방법에 따른 속도 비교

III-2절에서 제안된 레지스터 파일을 변형한 포워딩은 기존의 방법과 다른 버스 타이밍을 갖는다. 기존의 방법은 포워딩을 위한 지연 시간(delay time)이 포워딩 경로와 경로를 선택하기 위한 다중화기에 의해 결정되었다. 하지만 레지스터 파일을 사용하게 되면 레지스터 파일과 출력 버스를 추가로 거쳐야 한다.

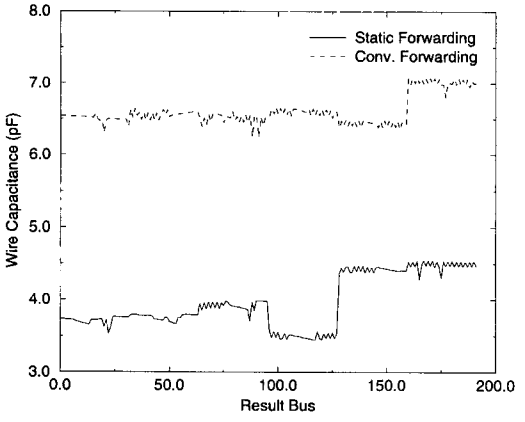


그림 11. 포워딩 경로의 정전용량
Fig. 11. Capacitance of forwarding paths.

포워딩 경로의 지연 시간은 충분한 최적화를 거친 후 비교해야 하지만 이것은 객관적 비교가 되기 어려우므로, 여기서는 동일한 조건하에서 설계된 두 가지 방법의 버스 정전용량(capacitance)을 측정하였다. 그림 11에서 볼 수 있듯이 기존의 포워딩은 평균 6.60pF이고 제안된 방법은 3.96pF으로 1.67배의 차이를 보인다. 이는 제안된 방법이 별도의 포워딩 경로를 필요로 하지 않기 때문에 버스에 적은 부하가 걸리는 것에 기인한다. 버스의 정전용량이 크면 버스에서의 신호 지연이 늘어나며 이를 보상해주기 위해서는 보다 많은 버퍼를 필요로 한다. 따라서 제안된 포워딩 경로의 지연이 기존 방법에 비해 작으며 효율적임을 알 수 있다.

레지스터 파일에 의한 지연 시간은 고속 센스 앰프를 사용함으로써 줄일 수 있다. 레지스터 파일에 의한 지연 시간이 포워딩 경로에서의 지연 시간 단축 이내로 설계된다면 제안된 방법에 의한 속도저하는 발생하지 않는다.

3. 포워딩 방법에 따른 장단점 분석

기존의 포워딩은 하드웨어를 통해 이루어지므로 프로그램 수행시 발생할 수 있는 어떠한 경우에서도 올바른 수행을 보장할 수 있었다. 하지만 정적 포워딩에서는 앞서 살펴본 바와 같이 컴파일러가 이를 효율적으로 해결해 주어야 하는 책임이 따른다. 따라서 컴파일러는 이를 고려해 설계되어야 한다.

포워딩 경로를 절약하기 위해서 레지스터 파일의 입출력 버스와 공유할 경우 많은 면적을 절약할 수 있었다. 버스 공유로 인한 버스 타이밍에서 손실은 버스 지연 시간의 단축과 고속의 레지스터 파일의 사용으로 상쇄시킬 수 있다.

정적 포워딩을 제어하는 신호를 명령어에 추가하기 위해서 레지스터 어드레스의 일부를 할당하였다. VPM에서는 총 64개의 어드레스 중 56개만이 레지스터에 사용되며 나머지 8개는 정적 포워딩 제어 신호로 사용된다. 사용할 수 있는 레지스터가 줄어들게 되면 상대적으로 메모리 입출력이 늘어나 프로그램 수행이 느려질 가능성이 있다. 하지만 포워딩을 위한 제어 신호가 사용되는 레지스터 수의 10% 내외로 작으며, 대부분의 응용 프로그램이 6 IPC를 지원하는 VPM에서 56개로 충분하였다. 또한 정적 포워딩을 사용함으로써 명령어 스케줄링(instruction scheduling)시 얻어지는 다음과 같은 이점이 이를 보상해 준다.

프로그램에서는 계산의 중간 결과로서의 임시 레지스터 사용이 빈번하다. 정적 포워딩을 사용할 경우 포워딩 경로를 임시 레지스터로 명령어 상에서 할당할 수 있게 되므로, duration이 d 사이클[4] 이하인 임시

[4] II-2절 참조

레지스터는 별도의 레지스터 할당을 필요로 하지 않는다. 또한 기존의 포워딩에서는 RAW hazard 조건이 발생했을 경우, 강제적으로 포워딩이 일어난 데 비해서 선택적인 포워딩을 할 수 있으므로 컴파일러는 보다 높은 자유도를 가지고 명령어 스케줄링을 할 수 있다.

V. 결 론

정적 포워딩은 RAW hazard의 발생과 그에 따른 포워딩의 제어를 컴파일 단계에서 소프트웨어로 결정하고, 명령어를 통해 제어 신호를 전달함으로써 별도의 포워딩 제어 회로가 필요하지 않다. 또한 레지스터 파일을 통해 포워딩 경로를 공유함으로써 많은 면적을 절약할 수 있었다. 컴파일러가 예측할 수 없는 프로그램 수행상의 여러 상황에 대해서도 큰 오버헤드 없이 해결할 수 있다. 따라서, VLIW 프로세서에서 정적 포워딩을 사용함으로써 RAW hazard 발생으로 인한 프로세서의 성능 저하를 효율적으로 방지할 수 있다.

참 고 문 헌

[1] David W. Wall, "Limits of Instruction-Level Parallelism," in *4th International Conference on Architectural Support for Programming Languages and Operating Systems,* Santa Clara, CA, April 1991, pp. 176-188.

[2] M. Butler, T.Y. Yeh, Y. Patt, M. Alsup, H. Scales, and M. Shebanow, "Single Instruction Stream Parallelism Is Greater Than Two," in *Proceedings of 18th Annual International Symposium on Computer Architecture,* May 1991, vol. 19, pp. 276-286.

[3] Mike Johnson, *Superscalar Microprocessor Design,* Prentice Hall, 1991.

[4] P.S. Ahuja, D.W. Clark, and A. Rogers, "The Performance Impact of Incomplete Bypassing in Processor Pipelines," in *Proceedings of 28th Annual International Symposium on Microarchitecture,* 1995.

[5] John L. Hennessy and David A. Patterson, Computer Architecture: A *Quantitative Approach,* Morgan Kaufmann Publishers, San Francisco, CA, 1996.

[6] A. Abnous and N. Bagherzadeh, "Architectural Design and Analysis of a VLIW Processor," *Computers Electrical Engineering,* vol. 21, no. 2, pp. 119-142, November 1995.

[7] J.A. Fisher, "Trace Scheduling: A Technique for Global Microcode Compaction," *IEEE Transactions on Computers,* vol. 30, no. 7, pp. 478-490, July 1981.

[8] James E. Smith and Andrew R. Pleszkun, "Implementing Precise Interrupts in Pipelined Processors," *IEEE Transactions on Computers,* vol. 37, no. 5, pp. 562-574, May 1988.

저 자 소 개

朴 炯 俊(正會員)

1995년 한국과학기술원 전자공학과 학사. 1997년 한국과학기술원 전자공학과 석사. 1997년 ~ 현재 한국과학기술원 전자공학과 박사과정. 주관심분야는 DSP, 영상 처리, 저전력 설계

金 利 燮(正會員)

1982년 서울대학교 전자공학과 학사. 1986년 Stanford University 전자공학과 석사. 1990년 Stanford University 전자공학과 박사. 1990년 ~ 1993년 Toshiba Corporation 연구원. 1993년 ~ 현재 한국과학기술원 전자공학과 부교수. 주관심분야는 멀티미디어 VLSI 디자인, 신호처리 알고리즘 구현, 저전력 설계