that there is no bus conflict when processor accesses data and code concurrently. In pipelined processors, pipelines can be stalled due to branch instuctions. To reduce this penalty we adopt BTB in the prefetch unit.. The prefetch unit consists of two 32byte buffer pairs. When a branch insruction shows up in a program, the prefetch unit predicts the targer address with the help of BTB. Then,branch target is fetched in one of the two buffer pairs which is not used at that time. If a branch insruction appears again, the other prefetch buffer pair is used. Such algorithm is implemented in HDL and performance improvement is verified.

Ⅰ. 서 론

컴퓨터의 중앙연산장치인 마이크로프로세서는 처리 속도를 높이기 위해 끊임없이 노력되어 왔다. 1970년대 후반 한 사이클에 하나의 명령어를 처리할 수 있는 파이프라인(pipeline) 기법[1]을 RISC(Reduced Instruction Set Computer) 형태의 마이크로프로세서에서 채택하여 급속한 처리 속도의 향상을 가져왔다. CISC 머신인 Intel 계열의 마이크로프로세서에서는 i386[2]부터 이것을 채택하여 처리 속도의 향상을 가져왔다. 이어 1980년대 말부터는 한 사이클에 다수 개의 명령어를 처리할 수 있는 수퍼스칼라 방식[1]을 채택한 프로세서 발표되면서 1990년도에는 고성능 마이크로프로세서가 상용되고 있다. 수퍼스칼라 방식은 여러 개의 파이프라인을 마이크로프로세서 내에 둠으로써 최대 파이프라인의 갯수만큼 명령어를 동시에 읽어 와서 수행하는 방식을 말한다. RISC 형태의 마이크로프로세서는 명령어의 길이가 일정[3]하여 첫 번째와 두 번째 명령어의 시작을 정확히 알 수 있어서 수퍼스칼라 방식을 구현하기 쉽다. 그러나, CISC 형태는 명령어의 길이가 일정하지 않으므로 첫 번째 명령어는 IP(Instruction Pointer)로 알 수 있지만 두 번째 명령어 시작은 알 수 없다. 따라서 CISC 형태의 마이크로프로세서가 수퍼스칼라 방식을 도입하기 위해서는 새로운 구조의 프리페치 유닛과 알고리즘이 필요하다.

본 논문에서는 현재 전 세계에서 가장 널리 사용되고 있는 intel의 x86 구조와 호환성을 유지하면서 RISC 구조의 특징을 갖는 CRISC 형태의 수퍼스칼라 파이프라인 구조에 적합하고 기존의 파이프라인 흐름을 방해하는 요소를 제거한 프리페치 유닛을 제시한다. 이 프리페치 유닛은 한 사이클에 두 개의 명령어를 처리할 수 있도록 두 개의 파이프라인(A-파이프, B-파이프)으로 명령어를 각각 보내며 분기 명령어로 발생되는 파이프라인 멈춤[4]을 해결하는 하드웨어를 포함한다. 내보낸 두 개의 명령어가 쌍을 이뤄 양 파이프라인에서 동시에 처리될 수 있을지 없을지는 디코딩하는 과정에서 결정하도록 설계하였다.

Ⅱ. 수퍼스칼라 파이프라인

2개의 파이프라인을 가진 YS6 수퍼스칼라 마이크로프로세서(현재 연세대학교 아식 설계 공동 연구소에서 설계중인 32비트 수퍼스칼라 마이크로프로세서) 구조는 그림 1과 같고 이를 위해서 모든 자원(resource), 예를 들어 디코더, 메모리 주소 발생기[5], ALU 등이 2개씩 필요하다. 한 개의 파이프라인에서 생기는 명령어 사이의 종속성(dependency) 문제[1],[6] 뿐만 아니라, 파이프라인이 두 개이므로 쌍을 이루는 명령어 사이에도 종속성이 발생한다. 이런 종속성으로는 첫째로 쌍을 이루는 명령어가 같은 자원을 사용하려 할 때 생기는 자원 종속성(resource dependency)이 있다. 이는 마이크로프로세서가 칩의 면적과 비용을 감안해 완전히 파이프라인화 되지 않았기 때문에 발생한다. 설계중인 YS6 프로세서에서도 barrel shifter 같은 것은 양 파이프에 존재하지 않고 주된 파이프인 A-파이프에만 있다. 따라서 디코더가 명령어를 짝짓기 할 때 shift 명령어는 A-파이프로 들어가게 해야 한다. 둘째로 쌍을 이루는 명령어 사이의 데이터 종속성(data dependency)이 있다. 이 종류로는 RAW(Read-After-Write), WAW(Write-After-Write)가 있으며 이를 해결하는 방법은 디코드할 때 두 개의 명령어가 짝을 이룰 수 없도록 만들면 가능하다. 셋째로 제어 종속성(control dependency)이 있다. 이것은 분기 명령어로 인해 파이프라인의 흐름이 갑자기 바뀔 때 미처 파이프라인으로 분기 타겟을 가져오지 못했기 때문에 파이프라인이 멈추는 것을 말한다.[7] 성능 향상을 위해 이를 해결하는 것이 필수적인데 본 연구에서는 분기 타겟 버퍼를 사용하여 이 문제를 해결하였다.

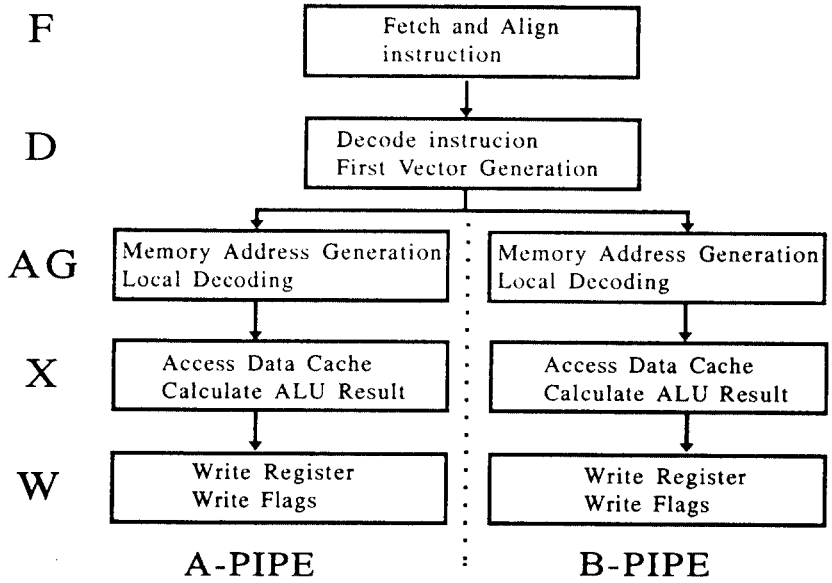

그림 1. YS6 파이프라인
Fig. 1. YS6 pipeline

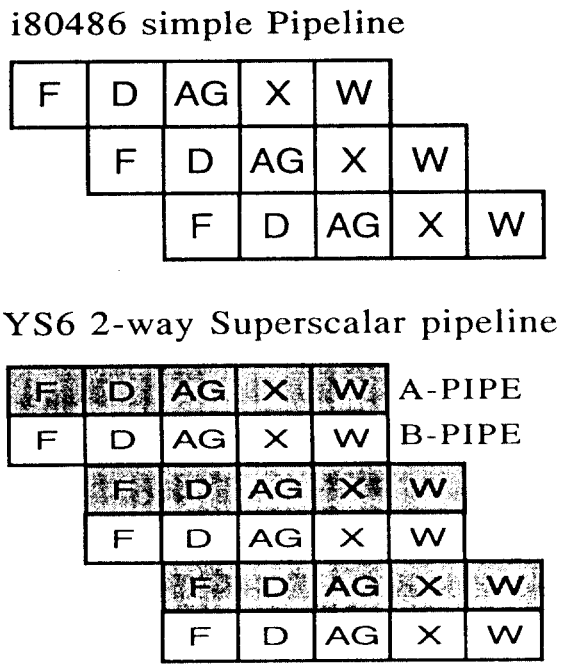

그림 2. 2-웨이 수퍼스칼라 파이프라인
Fig. 2. 2-way superscalar pipeline

파이프라인은 그림 2와 같이 5단계로 되어 있다.[8],[9] F 단계에서는 코드 캐쉬에서 32바이트 한 라인을 프리페치 버퍼로 가져오는 일을 한다. D 단계는 디코더가 버퍼에서 명령어를 받아 디코딩 한 후 종속성을 판단한다. AG 단계에서는 데이터 캐쉬 또는 메모리로부터 데이터를 가져오기 위해 메모리 선형 주소를 생성하며 제어유닛에서 나온 신호들을 각 유닛별로 재 디코딩을 한다. X 단계에서는 데이터 캐쉬를 엑세스 하거나 ALU 연산을 수행한다. W 단계에서는 레지스터 화일에 ALU 연산 결과를 저장하고 플래그 값들을 갱신하여 명령어의 수행을 마친다.

Ⅲ. 프리페치 유닛(Prefetch Unit)

1. 프리페치 유닛의 개요

프리페치 유닛은 코드 캐쉬나 메모리에서 명령어를 가져다가 수퍼스칼라 동작을 위해 두 개의 파이프라인으로 코드 캐쉬나 메모리에서 명령어를 공급하는 것이 기본적인 일이다. 이를 수행하기 위해 다음과 같은 블럭으로 구성된다. 최근에 수행되었던 명령어들을 담고 있는 코

드 캐쉬, 코드 캐쉬에서 수행해야 될 여러 가지 일 중에서 한 가지를 택해 코드 캐쉬로 선형 주소를 보내는 페치 제어 블럭, 선형 주소를 받아서 실제 주소로 바꿔주는 TLB, 수행될 명령어를 두 개의 파이프로 보내는 프리페치 버퍼, 수퍼스칼라 동작을 가능하게 하는 LBI 블럭, 분기 명령어를 파이프라인 멈춤 없이 처리할 수 있게 하는 BTB로 구성되며 전체 블럭 그림은 그림 3과 같다. 표1은 x86 계열의 프로세서에 관한 프리페치 버퍼(prefetch buffer)의 depth 변화 과정[10]을 보여주고 있는데, 프로세서의 처리 속도가 빨라짐에 따라 depth가 깊은 프리페치 버퍼가 필요하다.

표 1. 프리페치 버퍼의 깊이
Table 1. Prefetch buffer depth

프로세서	prefetch buffer의 depth(바이트)
286	6
386	16
486	32
pentium	two 64bytes queues
YS6	four 32bytes buffers

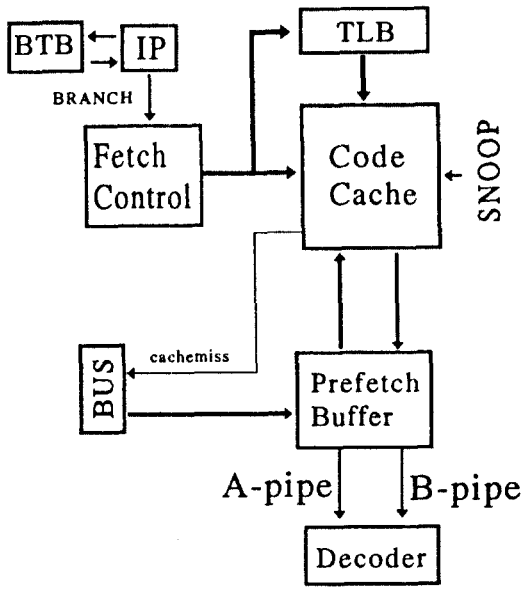

그림 3. 프리페치 유닛
Fig. 3. Prefetch unit

2. 분기 타겟 버퍼 블럭(Branch Target Buffer)

파이프라인의 정상적인 흐름을 방해하는 요소 중에서 보통 프로그램의 20%이상을 차지하는 분기 명령어가 있다.[11] 분기 명령어의 경우 제어 종속성으로 인한 성능 저하가 발생하는데, 타겟 주소를 계산해서 명령어를 가져오는데 시간이 걸리기 때문이다.[7] 설계중인 YS6 수퍼스칼라 마이크로프로세서는 분기 명령어를 처리하는 여러 알고리즘 중에서 분기 예측 알고리즘[11]을 사용하여 성능 저하를 줄일 수 있었다. 이것은 다음에 수행될 것으로 예측되는 명령어를 미리 가져와 실행을 한 뒤, 그 명령어가 X 단계에 도달하면 검증을 해서 파이프라인을 플러쉬(flush)시킬지 결정하는 것이다.

그림 4는 설계한 분기 타겟 버퍼이다. 4-way set associative 구조를 갖고 way당 64 엔트리(entry)를 가지며 모두 256 분기 명령어를 저장하도록 하였다. 분기 타겟 버퍼는 태그(tag), 타겟 주소, 예측정보, valid 비트, pseudo-LRU(Least Recently Used) 부분으로 되어 있다.

그림 5는 BTB를 이용하여 분기명령어를 처리하는 과정을 나타낸다. IP 블럭에서 현재 D 단계에 들어와 있는 명령어의 선형 주소를 받고 이 값은 BTB에 등록되어 있는 유효한 정보(valid bit=1)를 가진 태그와 비교하여 일치하면 history bit의 상태에 따라 분기를 할지 안할지 결정한다. 태그 부분은 이전에 실행된 분기 명령어의 바로 전 명령어의 선형주소가 등록되어 있으므로 나중에 똑같은 분기 명령어가 실행될 경우 태그 주소를 비교해 봄으로써 분기 명령어가 다음에 수행되는 것을 미리 알 수 있게 하였다. 이렇게 하는 이유는 분기 명령어의 선형주소가 태그에 들어 있게 되면 분기 타겟을 가져올 때 한 사이클의 지연이 발생하기 때문에 이전 명령어의 선형주소를 태그에 등록한다. 태그 부분과 비교되는 선형주소는 선형 명령어 포인터 모듈에서 발생되며 파이프라인에서 보면 D 단계에 비교하는 일을 한다. 타겟 주소 부분은 분기 명령어가 실행된다면 다음에 가져와 수행할 명령어의 주소를 저장하고 있다. 예측정보 부분에는 히스토리 상태 비트를 엔트리(entry)당 두 비트씩 할당하였다. 새로운 엔트리를 등록할 때는 항상 '11'로 초기화하고 상태 비트가 '11', '10', '01'인 경우는 분기를 취하고 '00'인 경우는 분기를 하지 않는 방안을 채택하였다.[12] 엔트리를 교체시키는 알고리즘은

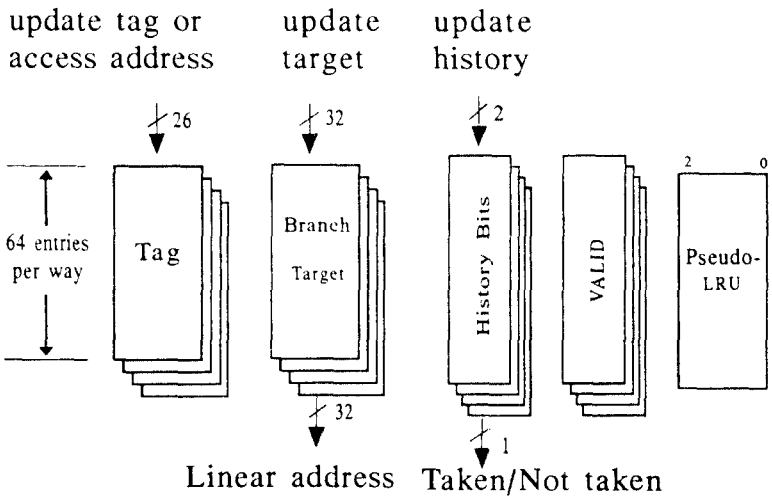

그림 4. 분기 타겟 버퍼 구조
Fig. 4. BTB structure

LRU에 비해 적은 비트를 사용하는 pseudo-LRU 방법을[12] 사용하여 3 비트를 할당하였다. Valid 정보 부분은 초기에는 '0' 이지만 엔트리가 할당될 때 '1'로 된다. 검증(verify)은 예측된 명령어가 X 단계에 들어갈 때 분기 명령어는 X 단계가 끝나게 되므로 분기 명령어에서 실제로 분기가 되었는지 또한 타겟이 어디인지 결정된다. 이 정보를 가지고 예측된 명령어의 선형주소와 비교하여 일치하면 그대로 흘러가게 하고 상태 비트

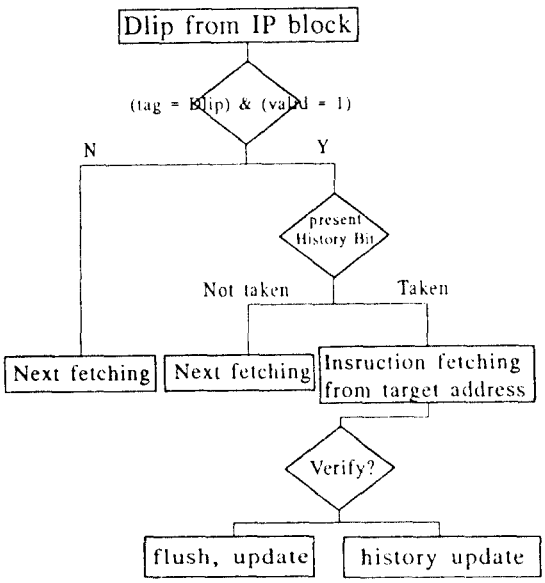

그림 5. 분기 예상의 흐름도
Fig. 5. Flow chart of branch prediction

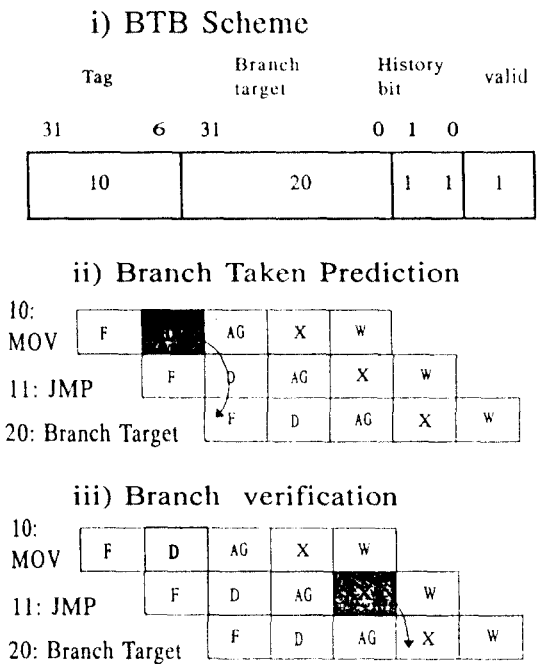

그림 6. 분기 명령어 처리 예
Fig. 6. Branch instruction process example

만 천이 시킨다. 틀리면 파이프라인을 플러쉬하고 BTB의 타겟과 상태 비트를 변경시킨다.

그림 6은 예를 들어 설명하는 것을 보여준다. BTB에 등록되어 있는 상태가 i)과 같고 파이프라인에 선형주소 10인 mov 명령어가 들어 있을 때 D 단계에서 BTB를 엑세스 한 결과 분기 수행으로 결정나 파이프라인 멈춤 없이 타겟 주소 20을 가져왔다. 선형주소 11의 jmp 명령어가 X단계를 끝내면 분기 타겟과 비교하는 과정을 수행한다.

3. 페치 제어 블럭

코드 캐쉬에서 수행하는 여러 가지 일은 다음과 같다.

① 플러쉬
② 한 라인 무효화
③ 한 라인 교체
④ 테스트 읽기
⑤ LBI(Last Byte Indicator) 수정
⑥ 읽기(분기 및 프리페치)

코드 캐쉬는 클럭(clock)당 한 가지 일만 처리할 수 있기 때문에 동시에 여러 가지 요청이 들어오게 되면 위의 순서로 우선권을 주며 우선권이 낮은 요청은 다음 사이클로 지연시키게 된다. 플러쉬 요청은 코드 캐쉬에 있는 내용을 모두 무효화시키는 일을 하며 우선적으로 수행해야 되는 중요한 일이기 때문에 우선권이 제일 높다. 한 라인 무효화는 특정 라인을 선택하여 무효화시키는 일이다. 한 라인 교체는 캐쉬 미스(miss)로 인해 버스를 통해서 외부 메모리로부터 코드를 읽어 와서 line fill buffer에 담았던 것을 코드 캐쉬로 쓰는 일이다. 테스트 읽기는 외부 유닛의 요청에 의해 캐쉬의 특정 라인을 테스트 레지스터로 쓰는 일이다. LBI 수정은 코드 캐쉬에 있는 명령어의 끝을 나타내는 LBI를 명령어 수행 후에 정확한 값으로 수정하는 일이다.

읽기 동작은 다음과 같이 두 가지가 있다.

① 분기 : 순차적인 파이프라인 흐름이 바뀌는 경우로 분기예측(branch prediction)과 분기 검증(branch verification)으로 이루어진다. 분기예측은 분기 타겟 버퍼에서 태그가 일치되고 예측 정보 비트의 상태에 의해 분기를 수행할 것인가를 결정하여 동작된다. 분기 검증은 분기 예상을 파이프라인 X 단계에서 검사하여 틀린 예상을 하였다면 정확한 분기 타겟으로 파이프라인 흐름을 바꾸게 된다. 분기 명령어 수행 뒤에는 항상 프리페치 읽기 동작을 곧바로 하게 만들어서 프리페치 버퍼를 비지 않게 만들었다.

② 프리페치 : 파이프라인 흐름을 바꾸지 않고 순차적으로 동작중인 캐쉬의 다음 라인을 프리페치 버퍼로 담는 일을 한다. 이 동작은 버퍼가 비어 있을 때나 분기 동작 후에 발생한다.

4. 코드 캐쉬 블럭

그림 7은 캐쉬를 코드와 데이터로 나누는 분리된 캐쉬 아키텍처(split cache architecture)와 합쳐진 캐쉬 아키텍처(combined cache architecture)를 비교한다.[13] 분리된 캐쉬 아키텍처는 주소 버스와 데이터 버스가 코드 캐쉬와 데이터 캐쉬에 따로 되어 있으므로 1 사이클에 동시에 엑세스가 가능하여 코드 읽기와 데이터 읽기의 충돌이 발생할 때 생기는 파이프라인 멈춤을 해결할 수 있고 캐쉬를 분리함으로써 생기는 설계상의 간단함(simplicity)을 얻을 수 있어서 캐쉬 구조는 분리된 캐쉬 아키텍처를 채택하였다.[14]

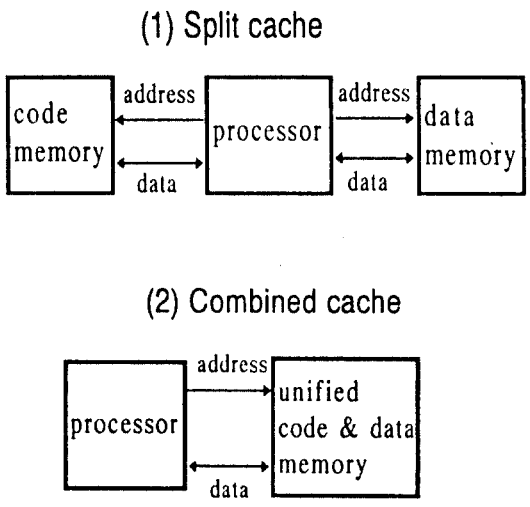

그림 7. 분리된 캐쉬와 합쳐진 캐쉬 구조
Fig. 7. Split cache vs. Combined cache structure

그림 8은 분리된 코드 캐쉬의 구조이다. 2-way set associative 방식으로 구성되며 크기는 16 Kbyte이

다. 256 sets가 있고 각 set은 2개의 라인을 갖고 있으며 각 캐쉬 라인은 32 바이트로 되어 있다. 코드 캐쉬 라인을 교체하는 알고리즘으로서 LRU 알고리즘을 사용했다. 2-way 방식이므로 LRU 비트는 '1' 비트로 구현이 가능하다. 즉 way0을 엑세스하면 LRU는 1로 되고 way1을 엑세스하면 LRU는 0으로 된다.

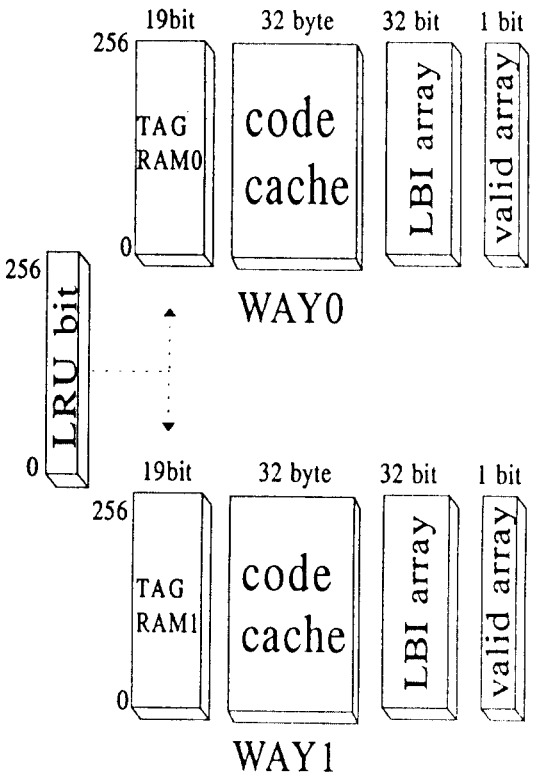

그림 8. YS6 코드 캐쉬 구조
Fig. 8. YS6 Code cache structure

캐쉬가 처음 동작할 때는 valid 비트가 모두 '0'인 invalid 상태이다. 캐쉬가 유효데이터를 가지고 있지 않은 상태에서 프리페치 요구가 들어오면 캐쉬 미스가 발생하여 버스 유닛으로 코드를 가져오도록 요청한다. 설계중인 버스의 폭(width)은 64비트로 캐쉬 미스(miss)가 발생된 주소가 캐쉬에 담을 수 있는 영역이면 버스는 burst mode로 동작하여 64비트씩 연속으로 4사이클에 걸쳐 32바이트를 가져와 line fill buffer에 담게 된다. 그 후 코드 캐쉬로 교체 요청 신호를 보내고 인지(acknowledge)신호를 받아 line fill buffer에 있는 내용을 캐쉬에 담는다. 데이터를 가져오는 방법으로 data-requested-last 방식과 data-requested-first[15] 방식이 있다. 본 연구에서는 data-requested-first을 채택하였다. 이 방식은 캐쉬 미스가 일어난 주소의 비트[4 : 3]의 값에 따라 첫번째 가져올 데이터의 주소를 결정한다. 그 주소에 해당하는 데이터는 line fill buffer 뿐만 아니라 즉시 마이크로프로세서에 전달되어 마이크로프로세서가 한 캐쉬 라인에 해당하는 4번의 전송(32 바이트)을 끝마칠 때까지 기다리는 것이 아니라 필요한 데이터를 받았기 때문에 다음 수행을 할 수 있다. 마이크로프로세서가 더 데이터를 요구하게 되면 line fill buffer를 엑세스하여 그 데이터를 line fill buffer에서 가져오게 된다. 이 방식을 이용하게 되면 line fill buffer의 추가와 제어 복잡성(control complexity)이 증가할지라도 파이프라인이 멈추는 사이클을 줄일 수 있는 장점 때문에 본 연구에서 구현하였다. 표 2는 캐쉬미스가 일어난 주소의 비트[4 : 3]의 값에 따라 line fill buffer에 담기는 순서를 나타내며 표 3은 그 값에 따라서 첫번째 가져올 주소를 결정하는 디코딩 표를 실었다.

표 2. 연속된 버스 전송 순서
Table 2. Burst bus transfer order

1st address	2nd address	3rd address	4th address
0	8	16	24
8	0	24	16
16	24	0	8
24	16	8	0

표 3. 비트[4:3] 디코딩 표
Table 3. bit[4:3] decoding table

비트[4 : 3]	1st address
00	0
01	8
10	16
11	24

성능 향상을 위해 4사이클이 끝난 뒤 디코드 유닛에

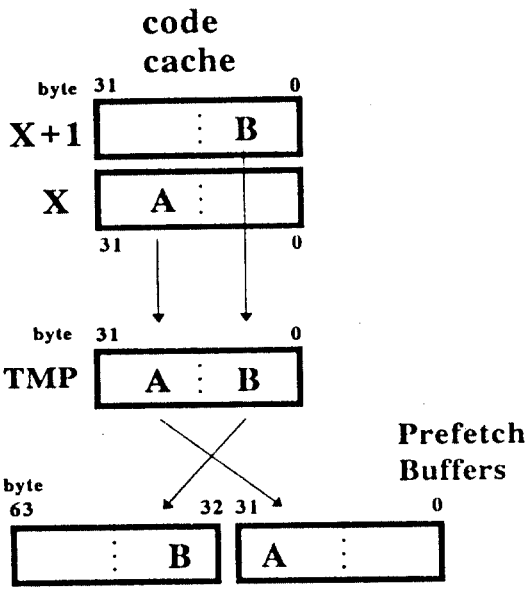

그림 9. 분리된 라인 동시에 가져오기
Fig. 9. Split line simultaneous fetch

명령어를 보내는 것이 아니라 가져오는 중간에 먼저 요청된 명령어를 내보내서 최대 3사이클의 이득을 얻을 수 있었다. 캐쉬 미스가 발생된 주소가 캐쉬에 담을 수 없는 영역이면 버스는 1사이클만 동작하여 8바이트만 보내게 된다. 이것은 마이크로프로세서에서 현재 코드 캐쉬의 동작 상태를 가리키는 신호를 버스로 내보내고 버스는 이 신호와 메모리에서 가져올 때 발생된 bus_fill 신호를 AND 하여 결정되도록 하였다. 즉 두 신호가 모두 로직 '1'이 될 때만 코드 캐쉬로 담을 수 있다.

코드 캐쉬의 태그는 3단자(triple port)로 엑세스 할 수 있다. 한 단자는 multiprocessing을 지원하기 위한 스누핑(snooping)용이고 나머지 두 단자는 다른 라인을 동시에 엑세스할 때 사용된다. 즉 현재 라인의 상위 16 바이트와 다음 라인의 하위 16 바이트를 가져올 때 필요하다. 이것은 명령어가 캐쉬의 현재 라인과 다음 라인에 걸쳐있을 때 동시에 가져옴으로써 1-15바이트까지의 가변 명령어 크기를 지원하는 마이크로프로세서에서 최악 조건에 대해서도 지연 없이 수행할 수 있기 때문에 구현한 방법이다. 위에서 설명한 내용이 그림 9에 있다. 만약 분기 타겟이 선형 주소 X인 라인의 상위 16바이트 중 어느 곳이라면 분기 타겟 라인 X의 상위 16바이트와 다음 라인 X+1의 하위 16바이트를 동시에 가져와서 그림 9와 같이 분기 명령어 전까지 동작 중이지 않던 프리페치 버퍼쌍 중의 하나로 입력된다. LBI 정보도 똑같은 과정으로 LBI만 처리하는 다른 버퍼 쌍으로 입력된

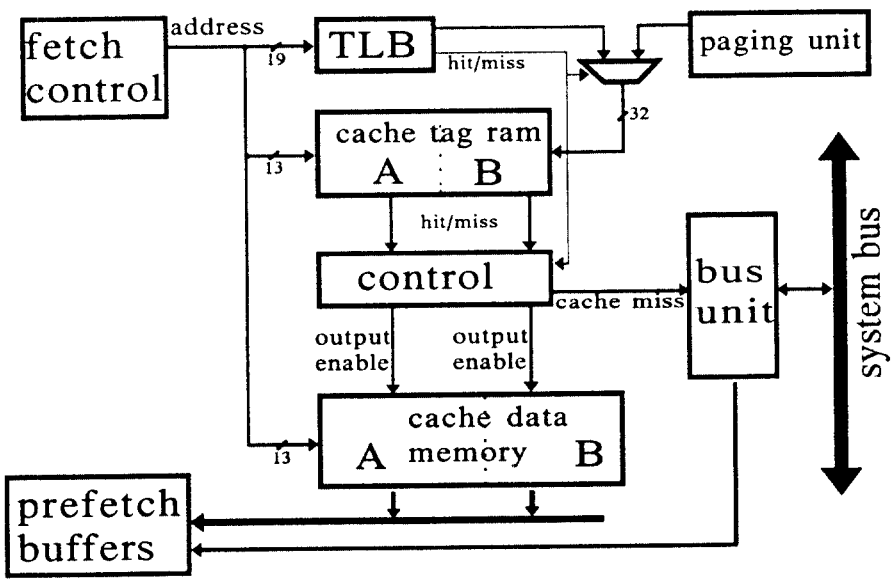

그림 10. 코드 캐쉬 엑세스
Fig. 10. Code cache access

다.

각 캐쉬는 실제 주소로 엑세스된다. 페치 제어 블럭에서 선형 주소 32 비트가 입력되면 상위 19 비트는 TLB에서 실제 주소로 번역되고 하위 13 비트 중에서 제일 아래 5 비트는 무시하고 나머지 8 비트로 태그램, LBI, LRU 비트, valid 비트를 읽어 낸다. 읽어낸 2개의 태그를 TLB에서 번역된 실제주소와 비교하여 캐쉬 히트(hit)인지 미스(miss)인지 결정한다. 양쪽에서 모두 미스가 발생하면 캐쉬 미스 요청을 버스로 보내고 히트가 발생하면 읽어낸 값을 프리페치 버퍼로 로드(load)한다. 이 과정은 그림 10에 있다.

5. 프리페치 버퍼 블럭

코드 캐쉬로부터 명령어들이 라인 단위로 프리페치 버퍼로 들어온다.[16] 따라서, 프리페치 버퍼는 32 바이트 크기가 된다. 그림 11에서 볼 수 있듯이 프리페치 버퍼는 32바이트 크기의 4개가 있고 2개씩 쌍을 이루고 있다. 이외에 LBI 정보를 받는 32비트 크기의 버퍼 4개가 있고 이것도 마찬가지로 2개씩 쌍으로 이루어져 프리페치 버퍼와 똑같은 동작을 한다. 프리페치 버퍼를 2개씩 쌍으로 한 것은 dynamic 동작을 위함이다. 이는 한 쌍이 동작중일 때 다른 한 쌍은 분기 명령어에 의한 파이프라인 흐름이 바뀔 때까지 쉬게 한다. 즉, 분기 명령어를 만날 때마다 동작하는 프리페치 버퍼 쌍이 변하게 된다. 또한 한 버퍼가 동작중일 때 미리 비어 있는 다른 버퍼로 명령어를 가져 올 수 있게 하려고 2개를 한 쌍으로 했다.

그림 11에서 두 번째 mux 선택은 분기 명령어에 의한 파이프라인 흐름을 바꿀 때 사용한다. 그 옆의 rotator는 명령어 포인터에 따라서 D 단계에 해당하는 명령어 디코더로 양 파이프, 즉 A 파이프와 B 파이프에 필요한 명령어를 공급한다. 동시에 LBI 정보를 보내준다.

LBI는 설계중인 YS6 수퍼스칼라 마이크로프로세서가 두 개의 명령어를 동시에 처리할 수 있게 하는 결정적인 정보이다. 첫 번째 명령어의 시작은 읽기 포인터(Read Pointer)에 의해 항상 알 수 있다. 두 번째 명령어의 시작은 LBI array값(A-파이프로 보내는 11바이트에 해당하는 11비트 정보) 중에서 '1' 다음부터로 정한다. 즉 '1'이면 첫 번째 명령어의 끝임을 나타내도록 하여 LBI 정보가 옳다면 두 번째 명령어의 시작을

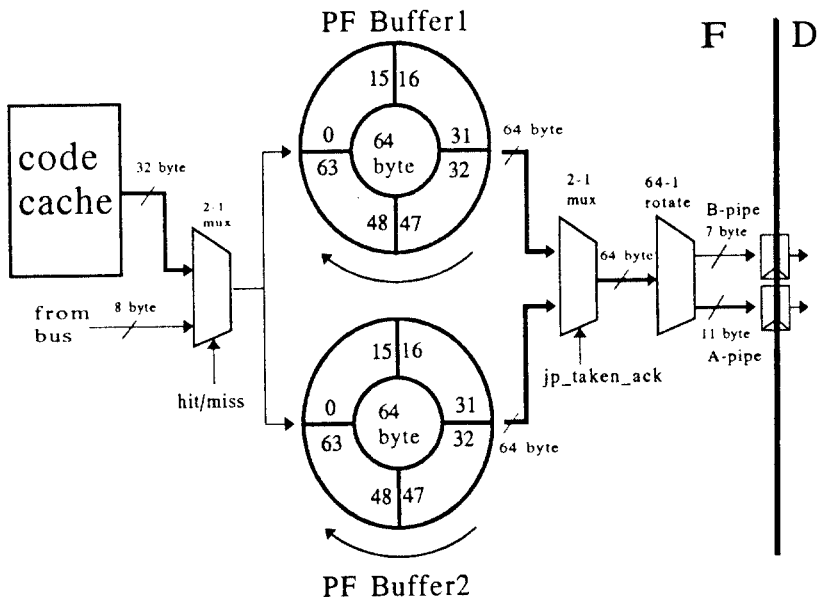

그림 11. 프리페치 버퍼 구조
Fig. 11. Prefetch buffer structure

알 수 있으므로 두 개의 명령어 처리가 가능하며 틀리면 디코더는 첫 번째 명령어만 A-파이프를 통해 수행된다. 한번도 실행된 적 없는 명령어는 당연히 캐쉬 미스가 발생되어 버스를 통해 들어오며 이때 LBI는 모두 '1'로 초기화된다. 즉 이 때는 우연히 명령어가 1바이트이면 두 개가 수행될 수 있지만 대부분 한 개의 명령어만 수행되고 각 명령어가 디코더를 통과하면 몇 바이트의 명령어인지 알 수 있으므로 이 정보를 통해 LBI를 update 시키고 line fill buffer에서 32바이트의 코드를 캐쉬로 쓸 때 update된 이 32비트 LBI 정보도 같이 쓰여진다. 그러면 다음에 이 라인을 다시 읽을 때 정확한 값을 얻을 수 있고 또한 B-파이프로 두 번째 명령어를 보내는 것이 가능하다. 디코더는 명령어를 디코딩 한 뒤 프리페치 유닛에서 받은 LBI array 값을 비교하여 수정하라는 신호를 프리페치 유닛으로 보낸다. 한 라인에 해당하는 정보를 수정한 뒤 페치 제어 블럭으로 LBI 수정 요청 신호를 보낸다.

그림 12는 읽기 포인터를 만드는 것이다. 프로그램이 실행될 때 첫 번째 시작의 명령어 주소는 IP로부터 받고 이 값의 선형 주소[5 : 0]가 맨 처음의 읽기 포인터로 선택되고 그 다음 클럭에는 두 개의 명령어가 쌍을 이뤘는가 못 이뤘는가에 따라서 이전 읽기 포인터에 tot_length(A-파이프 와 B-파이프로 들어간 명령어의 총 길이)가 더해질지 a_length(A-파이프로 들어간 명령어의 길이)만 더해질지 결정된다. 프리페치 버퍼는 이 값을 받아서 프리페치 버퍼로부터 A-파이프로 들어갈 명령어와 LBI의 도움으로 B-파이프로 들어갈 명령어를 선택한다. 프로그램 실행 중간에 분기를 만나게 되면 mux의 선택을 통해 읽기 포인터를 변경시킬 수 있다.

Ⅳ. 시뮬레이션 및 결과

본 논문에서는 프리페치 unit을 Verilog HDL[17]을 사용하여 동작 시뮬레이션을 수행하였다. 먼저, 시뮬레이션의 복잡도를 줄이기 위해서 그림 3에서 보인 전체 블럭을 여러 개의 모듈로 나누고 각 모듈을 상호 연결하는 계층적 구조 방식을 사용하였다.

그림 13은 코드 캐쉬가 수행할 여러 가지 일 중에서 우선권에 의해서 한 가지 일만 입력되는 것을 보여준다. labus는 코드 캐쉬로 입력되는 선형주소 버스이며 신호 이름 뒤에 req가 붙은 것은 코드 캐쉬에서 수행해야 될 일을 요청하는 신호를 나타낸다. 그림에서 보면 #51에서 lbireq와 prereq가 동시에 요청될 때 우선권에 의해 lbireq가 먼저 처리되고 지연된 prereq가 다음 사

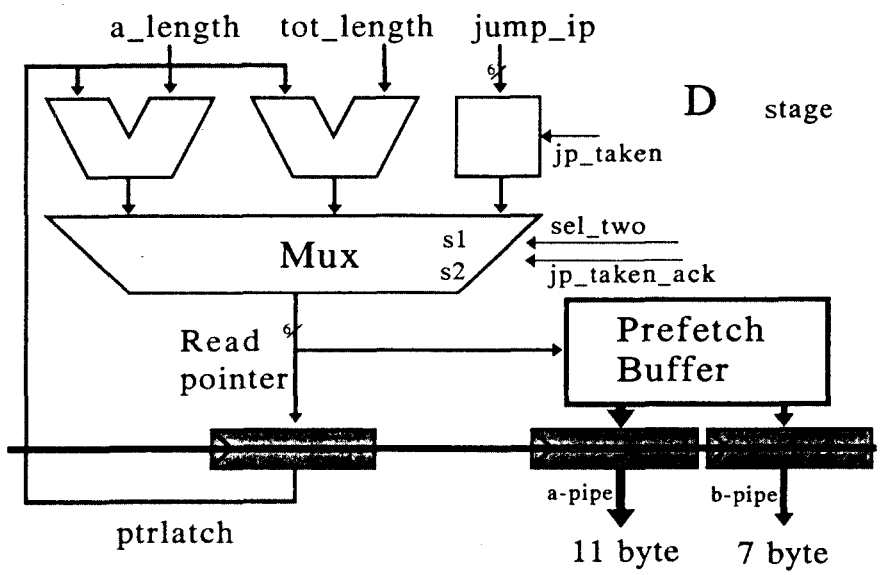

그림 12. 읽기 포인터 만드는 과정
Fig. 12. Read pointer generation

이클에 처리된다.

그림 14는 코드 캐쉬의 동작을 보여준다. brareq (branch request)를 입력으로 받아서 splbra 신호를 만들고 split branch를 수행한다. #48에 codout에 실린 code가 이것을 나타낸다. #8에서는 brareq에 의한 분기 동작으로 codout에 실제 주소 0001af0f에 있는 코드를 실어서 다음 사이클에 파이프라인의 멈춤 없이 분기 타겟을 디코딩 할 수 있도록 프리페치 버퍼로 담게 된다. #28에서는 프리페치 동작으로 실제 주소 0001af2f에 있는 코드가 codout에 실렸음을 보여준다.

그림 15는 읽기 포인터가 어떻게 변화되는가를 보인다. 첫 번째 rd_ptr은 분기로 인한 값 15를 받는다. 두 번째 rd_ptr은 디코딩이 끝난 후 두 개의 명령어를 처리했으므로(two=1) A-파이프 길이와 B-파이프 길이를 더한 값 26으로 다음에 프리페치 버퍼에서 읽어 올 곳을 나타낸다. LBI는 A-파이프로 보내는 11바이트 중에서 6번째 비트가 '1'이므로 실제 디코딩한 값 즉, A-파이프 opcode가 2바이트이고 immediate 값이 4바이트이므로 합하면 6바이트가 되므로 LBI가 나타낸 값과 같고 이를 근거로 두 개의 명령어를 처리할 수 있음을 나타낸다. 세 번째 rd_ptr은 LBI가 틀린 값을 나타내므로 두 개의 명령어를 처리 못하고 A-파이프 길이 11만큼 증가한다. Prereq(prefetch request)는 rd_ptr이 32를 넘어설 때 assert되며 이것은 프리페치 버퍼가 하나 비었다는 것을 페치 제어 블럭에 알려주는 신호이

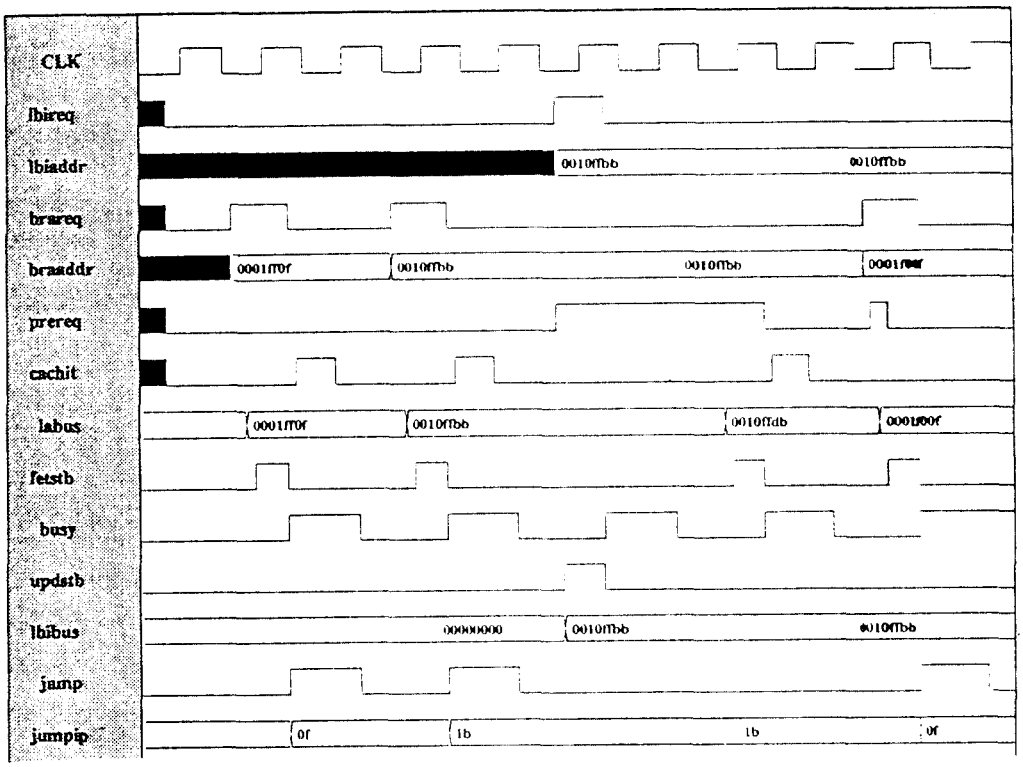

그림 13. 페치 제어 시뮬레이션
Fig. 13. Fetch control simulation

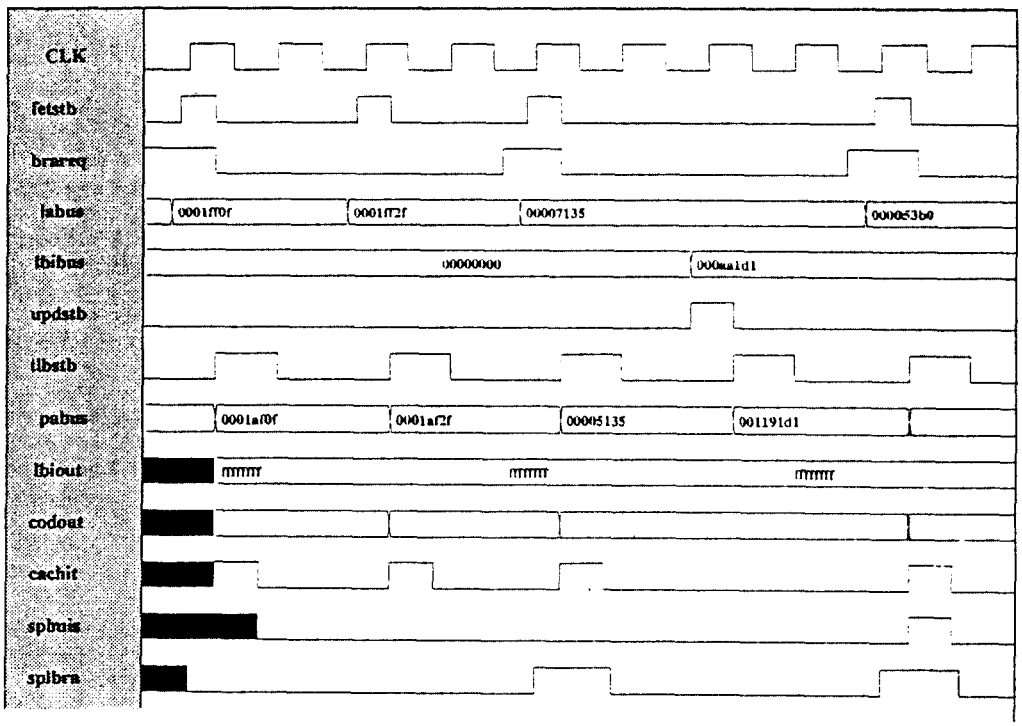

그림 14. 코드 캐쉬 시뮬레이션
Fig. 14. Code cache simulation

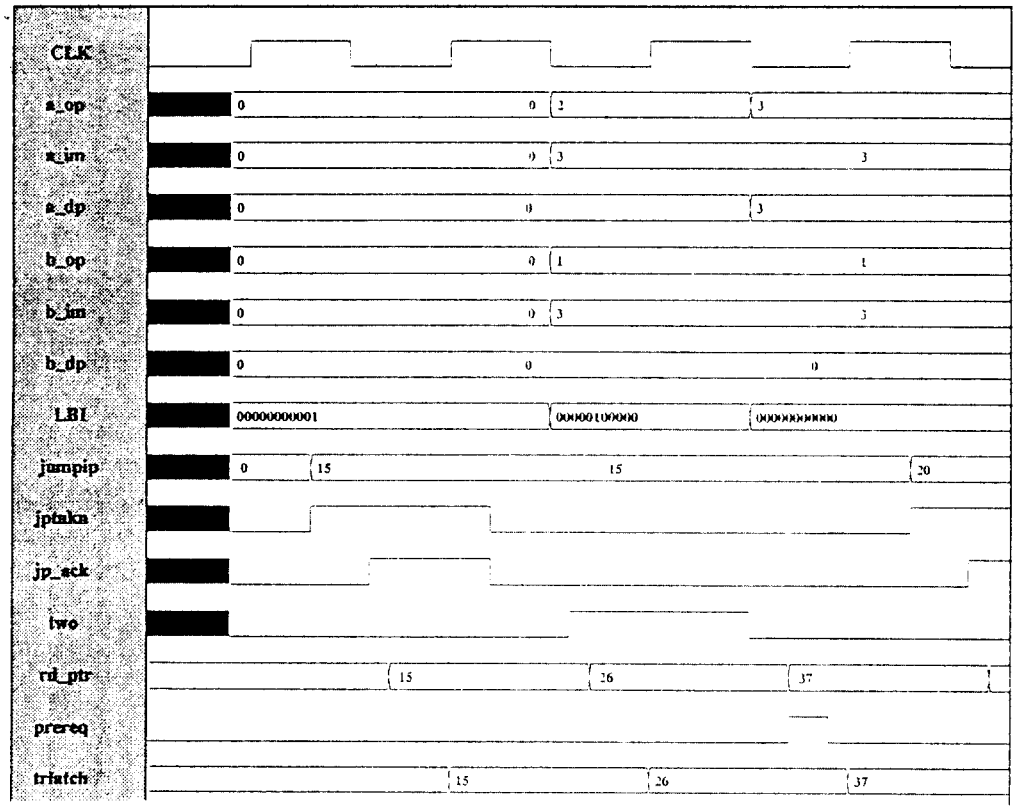

그림 15. 읽기 포인터 발생 시뮬레이션
Fig. 15. Read pointer generation simulation

다.

그림 16은 프리페치 버퍼에서 rd_ptr값으로 A-파이프와 B-파이프로 명령어를 내보내는 것을 나타낸다. 분기 타겟으로 rd_ptr이 39로 된 다음 A-파이프로 거기서부터 11바이트 그리고 LBI 정보로 7바이트가 지난 뒤부터 즉 opcode가 88로 시작되는 곳부터 B-파이프로 7바이트를 보낸다. 다음 클럭에 디코더는 이것을 디코딩한 후 두 개를 처리한 결과 rd_ptr을 50으로 변화시킨다. 다시 여기서부터 A-파이프로 11바이트, LBI가 첫 번째에서 '1'이므로 B-파이프로 opcode가 44곳부터 7바이트가 입력된다.

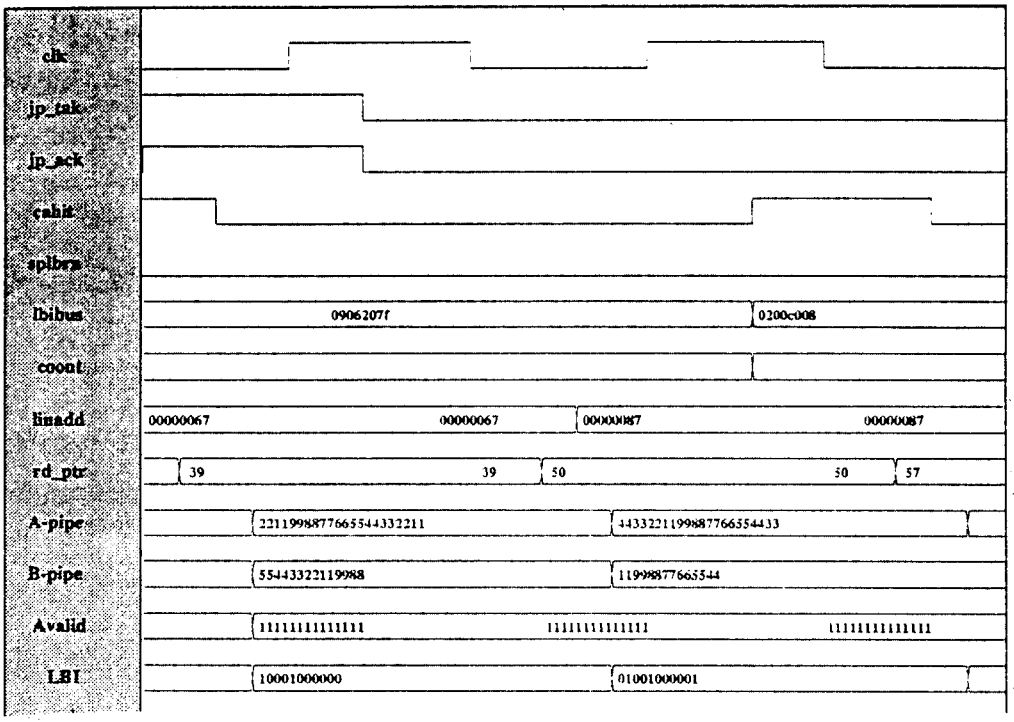

그림 16. 프리페치 버퍼 시뮬레이션
Fig. 16. Prefetch buffer simulation

Ⅴ. 결 론

본 논문에서 한 사이클에 두 개의 명령어를 수행할 수 있는 수퍼스칼라 YS6 마이크로프로세서의 프리페치 unit의 아키텍처를 정의하였고 Verilog HDL을 사용하여 기능수준에서 이의 동작을 검증하였다.

한 사이클에 2개의 명령어를 수행하기 위해서는 두 번째 명령어의 시작 위치를 알아야 하는 것이 무엇보다 중요하다. 이를 위해 본 논문에서 LBI를 사용하여 해결하였다. 또한 분기 명령어로 인해서 파이프라인 흐름이 정지되어 3 사이클이 지연되는 것을 막기 위해 BTB를 사용하여 성능 향상을 가져올 수 있었다. 두 개 쌍의 프리페치 버퍼를 두어 분기 명령에 대해 효율적으로 동작하도록 하고 한 쌍의 프리페치 버퍼를 32바이트 크기의

두 개의 버퍼로 구성하여 어느 한 쪽이 비면 곧바로 프리페치를 요구하도록 하여 파이프라인이 멈추는 것을 최소화하였고 명령어가 코드 캐쉬에서 두 개 라인에 걸쳐 있을때 역시 한 사이클에 프리페치가 가능한 알고리즘을 구현하였다.

본 논문에서 2-way 수퍼스칼라 마이크로프로세서에 적합한 구조를 갖는 프리페치 유닛을 보였는데 여기에 별도의 하드웨어 추가없이 3-way, 4-way를 갖는 수퍼스칼라 마이크로프로세서에 맞도록 수정이 가능하며 앞으로 연구 방향도 차세대 마이크로프로세서인 686이나 786프로세서에 적합한 프리페치 유닛 설계가 될 것이다.

Ⅵ. 參考文獻

1. Mike Johnson, *Superscalar Microprocessor Design*, Prentice Hall, New Jersey, p.1-7, pp.9-30, 1991.
2. Chris H. Pappas & William H. Murray, *80386 Microprocessor handbook*, McGraw-Hill, Berkely, pp.20-24, 1988.
3. Kai Hwang, *Advanced computer architecture*, McGraw-Hill, pp.157-177, 1993.
4. Honesty C. Young, Eugene J. Shekita, "An intelligent I-cache Prefetch Machanism", *ICCD*, pp.44-49, 1993.
5. 정대석, 고성능 마이크로프로세서 선형주소 생성기 HDL 모델링, 연세대학교 석사학위 논문, 1994.
6. John L Hennessy & David A Patterson, *Computer architecture a quantative approach*, Morgan Kaufmann Publishers, San Mateo CA., pp.252-278, 1990.
7. David J. Lilja, "Reducing the Branch Penalty in Pipelined processors", *IEEE computer*, pp.47-55, 1988.7.
8. Beatrice Fu, Avtar Saini, Patrick P. Gelsinger, "Performance and microarchitecture of the i486TM pocessor", *ICCD*, pp.182-187, 1989.
9. Brain Case, "Intel reveals Pentium implementation Details", *Microprocessor Report*, Vol. 7, No. 14, 1993.10.
10. Don Anderson & Tom Shanley, *Pentium Processor System Architecture*, MindShare Press, Texas, p.19, pp.135-164, 1993.
11. Johnny K.F., Lee Alan Jay Smith, "Branch Prediction Strategies and Branch Target Buffer Design", *IEEE Computer*, pp.6-21, Jan. 1984.
12. 장은숙, 고성능 마이크로프로세서 YS6의 분기 타겟 버퍼의 HDL 모델링, 연세대학교 석사학위 논문, 1994.
13. Jim Handy, *The cache memory book*, Academic press, San Diego, pp.107-117, 1993.
14. Rodney Boleyn, James Debradelabin, Vivek Tiwari, Andrew Wolfe, "A Split Data Cache for Superscalar Processors", *ICCD*, pp.32-37, 1993.
15. Intel, *Pentium Processor User's Manual*, Vol. I, 6.13-6.14, 1993.
16. F. Arakawa, K. Uchiyama, et al, "A CMOS 50MHz CISC Superscalar Microprocessor", *ISSCC*, pp.11-12, 1993.
17. Eli Sternheim, Rajvir Singh, et al, Digital Design and Synthesis with *Verilog HDL*, Design automation series, 1993.5.

李 鎔 錫 (Yong Surk Lee) 정회원
1950년 10월 23일생
1969년 3월~1973년 2월 : 연세대학교 전기공학과(공학사)
1975년 3월~1977년 2월 : Michigan 반도체설계 공학석사
1977년 3월~1981년 12월 : Michigan 반도체설계 Ph.D
1982년 3월~1983년 12월 : Sperry-Univac Comput 설계 엔지니어
1984년 1월~1986년 9월 : Hyundai Electronics, USA 설계 엔지니어
1986년 10월~1988년 8월 : National Semiconductor 설계 엔지니어
1988년 9월~1989년 8월 : Performance Semiconductor 설계 엔지니어
1989년 9월~1992년 12월 : Intel Corporation 설계 엔지니어
1993년 3월~현재 : 연세대학교 전자공학과 부교수