64 비트 4 - way 수퍼스칼라 마이크로프로세서의 최적화된 명령어 이슈 구조

연세대학교 대학원

전자공학과

문 병 인

64 비트 4 - way 수퍼스칼라 마이크로프로세서의 최적화된 명령어 이슈 구조

지도 이 용 석 교수

이 논문을 석사 학위논문으로 제출함

1996년 12월 일

연세대학교 대학원

전자공학과

문 병 인

문병인의 석사 학위논문을 인준함

심사위원 ______________________ 인

심사위원 ______________________ 인

심사위원 ______________________ 인

연세대학교 대학원

1996년 12월 일

감사의 글

부족한 이 한편의 논문을 마치면서 지난 대학원 생활 동안 저에게 도움을 주신 분들께 진심으로 감사 드립니다. 그 동안 특별한 관심으로 공부를 지도해 주시고 앞으로 나아갈 방향을 충고해 주신 이용석 교수님의 은혜에 감사 드립니다. 항상 자상한 배려와 인자하신 마음으로 지도와 격려를 하여 주신 김봉렬 교수님과 이문기 교수님께 감사 드립니다. 또한 심사 과정에서 부족한 내용을 자상하게 지적해 주신 김재석 교수님께 감사 드립니다.

학부와 대학원 과정 동안 친절한 가르침으로 지도하여 주신 박규태 교수님, 차일환 교수님, 이상배 교수님, 강창언 교수님, 박한규 교수님, 박민용 교수님, 윤대희 교수님, 김재희 교수님, 이재용 교수님, 홍대식 교수님, 최우영 교수님, 송홍엽 교수님, 이철희 교수님께 진심으로 감사 드립니다.

대학원 생활 동안 같은 분야의 연구를 하면서 자상한 배려와 다정한 조언으로 보살펴 주신 이용한 선배님과 안종근 선배님께 감사 드립니다. 또한 회사에 가서도 아낌없는 사랑과 도움을 주신 이원 선배님께 감사 드립니다. 그리고 연구실 생활을 충실하게 할 수 있도록 도와주신 서광수, 이광엽, 최병윤, 이승호, 남승현, 손승일, 정갑중, 김재영, 이보나, 정한힘, 안상준, 김근회, 정태식, 이태영, 이종익, 오승호, 곽승호, 양훈모, 김경환 선배님과 호길 형, 영완 형, 상욱 형, 동수 형에게 감사의 마음을 전합니다. 아울러 연구실 생활을 함께 했던 동환 형, 한준 형, 준환 형, 상국, 승우,

권경환, 용규, 홍철, 성우, 대옥, 정일, 우경, 호경, 영덕, 용운, 종수, 한상, 영록, 성주 형, 곽재영, 재욱, 자용 형, 상오에게도 감사의 뜻을 전합니다.

10년이 넘는 세월 동안 사랑과 우정으로 기쁨과 힘을 준 재영, 은영, 우진, 상진, 상학, 영훈, 완수, 충남, 범이에게 진심으로 감사의 마음을 전합니다. 그리고 윤성에게 특별한 고마움의 뜻을 전합니다.

저에게 기쁜 일이 있을 때는 같이 기뻐하시고 어려운 일이 있을 때는 해쳐나갈 수 있도록 도와주신 저의 소중한 아버지, 어머니께 사랑과 감사의 마음을 드립니다. 끝으로 형, 큰 누나, 작은 누나, 형수님, 큰 매형, 작은 매형, 꼬마 족하들과 이 작은 기쁨을 나누고자 합니다.

1996년 12월

문 병 인

차 례

감사의 글

그 림 차 례 ······ iii

표 차 례 ······ v

국 문 요 약 ······ vi

제 1 장 서 론 ······ 1

제 2 장 마이크로프로세서 구조 ······ 4

2.1 절 4-way 수퍼스칼라 마이크로프로세서 ······ 4

2.2 절 이슈 유닛 ······ 11

제 3 장 이슈 유닛의 설계 ······ 14

3.1 절 명령어 정렬기 ······ 14

3.1.1 절 명령어 정렬기의 동작 ······ 14

3.1.2 절 명령어 정렬기의 구조 ······ 18

3.2 절 명령어 버퍼 ······ 21

3.2.1 절 명령어 버퍼의 동작 ······ 21

3.2.2 절 명령어 버퍼의 구조 ······ 24

3.3 절 그룹화 로직 ······ 31

3.3.1 절 그룹화 로직의 동작 ······ 31

3.3.2 절 그룹화 규칙 ······ 32

3.3.3 절 그룹화 로직의 구조 ······ 49

제 4 장 모의실험 및 검증 ······ 57

4.1 절 하위 블럭의 검증 57
4.2 절 이슈 유닛의 검증 60
제 5 장 결론 65
참고 문헌 67
부 록 70
영 문 요 약 80

그 림 차 례

그림 2-1 4-way 수퍼스칼라 마이크로프로세서의 전체 블럭다이어그램 ···· 7

그림 2-2 4-way 수퍼스칼라 마이크로프로세서의 9 단 파이프라인 ············ 8

그림 2-3 이슈 유닛의 블럭다이어그램 ············ 12

그림 3-1 명령어 정렬기의 입출력 관계의 한 예 ············ 17

그림 3-2 첫째 명령어가 무효한 경우 ············ 17

그림 3-3 명령어 정렬기의 구조 ············ 19

그림 3-4 멀티플렉서 선택 신호 발생 로직 ············ 20

그림 3-5 출력 유효 비트의 발생 로직 ············ 20

그림 3-6 명령어 정렬과 명령어 이슈 타이밍 ············ 23

그림 3-7 명령어 버퍼의 블럭다이어그램 ············ 25

그림 3-8 head pointer 의 발생 ············ 26

그림 3-9 tail pointer 의 발생 ············ 27

그림 3-10 정수 명령어의 true dependency ············ 37

그림 3-11 정수 명령어의 output dependency ············ 37

그림 3-12 정수 기능 유닛의 충돌 ············ 38

그림 3-13 분기 명령어와 주변 명령어들의 이슈 ············ 41

그림 3-14 load 명령어에 대한 데이터 의존성 ············ 43

그림 3-15 부동소수점/그래픽 명령어의 데이터 의존성 ············ 47

그림 3-16 그룹화 로직의 블럭다이어그램 ············ 50

그림 3-17 첫째 명령어를 이슈하는 블럭의 구조 ············ 53

그림 3-18 정수 명령어 데이터 의존성 검사 로직 55
그림 4-1 하위 블럭의 검증 59
그림 4-2 C 프로그램을 이용한 이슈 유닛의 검증 62
그림 4-3 성능 평가 과정 63

표 차 례

표 2-1 4-way 수퍼스칼라 RISC 마이크로프로세서의 사양 ·············· 6
표 3-1 명령어 정렬기의 입출력 관계 ·············· 16
표 3-2 정수 명령어의 분류 ·············· 34
표 3-3 다중 사이클 명령어와 bubble 수 ·············· 35
표 3-4 부동소수점/그래픽 명령어의 분류 ·············· 45
표 3-5 4 배정도 부동소수점 명령어와 관련 트랩 ·············· 48
표 3-6 명령어 해석 블럭과 해석 명령어 종류 ·············· 54
표 4-1 C 프로그램 수행 결과 ·············· 61

국 문 요 약

64 비트 4-way 수퍼스칼라 마이크로프로세서의 최적화된 이슈 구조

본 논문은 4-way 수퍼스칼라 RISC 마이크로프로세서의 구조에서 명령어 저장과 이슈를 담당하는 이슈 유닛의 설계 및 검증에 관하여 기술한다. 이슈 유닛은 SPARC version 9 의 명령어 세트를 사용하여 설계되었다. 또한 그래픽 처리를 위한 영상 명령어 세트(Visual Instruction Set, VIS)를 추가하여 영상 처리 프로그램을 효율적으로 수행할 수 있도록 하였다. 이슈 유닛은 프리페치 유닛에서 명령어를 제공받으며, 단일 사이클에 최대 4 개의 명령어를 순차적 방식으로 이슈하여 기능 유닛들을 최대한 활용하도록 하였다. 또한 이슈 유닛은 명령어 버퍼를 사용하여 명령어의 페치와 이슈를 분리시킴으로써, 명령어의 이슈율(instruction issue rate)을 높였다.

이슈 유닛은 명령어 정렬기(instruction aligner), 명령어 버퍼(instruction buffer) 및 그룹화 로직(grouping logic)으로 구성되었다. 명령어 정렬기는 입력된 4 개의 명령어 중에서 처음으로 유효한 명령어가 명령어 버퍼의 첫째 입력 명령어가 되도록 순서를 조정하는 역할을 한다. 명령어 버퍼는 명령어 정렬기에서 입력된 명령어를 정렬하여 저장함으로써 그룹화 로직에 충분한 명령어를 제공하는 기능을 담당한다. 그룹화 로직은 명령어 버퍼에서 입력받은 명령어를 해석하고, 실행 유닛과 레지스터파일의 사용 가능성 및 데이터 의존성을 고려하여 실행 유닛으로 이슈하는 기능을 한다.

이슈 유닛은 behavioral 수준 및 structural 수준에서 HDL(Hardware Description Language)을 이용하여 기술되었다. 또한 하위 수준의 설계를 고려하여 이슈 유닛의 HDL 모델은 자동합성(synthesis)이 가능하도록 만들어졌다. 명령어 캐쉬 및 데이터 캐쉬의 히트율(hit rate)과 분기 예측의 정확성을 100%로 가정하였을 때 C 프로그램을 대상으로 모의실험(simulation)을 수행한 결과 1.8의 IPC(Instruction Per Cycle)를 기록하였다. 그러나 최적화된 컴파일러에 의해서 명령어 스케쥴링(instruction scheduling)이 수행된 프로그램을 사용한다면 더 좋은 성능을 가질 것이다.

핵심되는 말 : 4-way 수퍼스칼라, RISC 마이크로프로세서, 이슈 유닛,
명령어 이슈율, 명령어 정렬기, 명령어 버퍼, 그룹화 로직

제 1 장 서 론

명령어 세트와 컴파일러의 설계 분야에서 최근 보여준 발전은 고성능의 마이크로프로세서 아키텍처의 개발로 이어졌다. 특히 CISC(Complex Instruction Set Computer) 방식에서 RISC(Reduced Instruction Set Computer) 방식으로 아키텍처가 발전하면서 대부분의 명령어를 단일 사이클에 실행할 수 있게 되었으며 반도체 공정 기술의 발달로 인해 클럭 사이클 시간을 줄임으로써 마이크로프로세서의 성능은 크게 향상되었다. 그러나 정보화 사회가 진전됨에 따라 점점 더 대용량화 되어가는 응용 프로그램과 데이터를 효율적으로 지원하기 위해서 새로운 고성능 마이크로프로세서의 개발이 지속적으로 필요하게 되었다. 이러한 요구에 따라 현재 많이 연구되고 있는 아키텍처[1] 중에는 수퍼스칼라 마이크로프로세서[2], VLIW(Very Long Instruction Word)[2-4], 멀티쓰레디드(multithreaded) 방식[5], 멀티프로세서 방식[6] 및 네트워크 컴퓨터 등이 있다. 이 중에서 수퍼스칼라 마이크로프로세서는 기존의 응용 프로그램과 호환성을 유지하면서 명령어 수준 평행성(instruction level parallelism)을 이용하여 성능 향상을 꾀할 수 있다는 장점을 가지고 있다.

수퍼스칼라 마이크로프로세서에서 성능에 가장 큰 영향을 미치며 제일 복잡한 부분이 페치된 명령어들을 실행 유닛들에 이슈하는 유닛이다. 본 논문의 이슈 유닛은 명령어 세트(instruction set)를 SPARC-V9[7,8]으로 하는 64 비트 4-way 수퍼스칼라 RISC 마이크로프로세서의 이슈 유닛이다. 현재 SPARC-V9 을 사용한 4-way 수퍼스칼라 마이크로프로세서에는 Hal Computer

Systems 의 SPARC64[9], Sun 의 UltraSPARC-I[10-14] 등이 있다.

본 논문의 수퍼스칼라 마이크로프로세서는 순차적 방식으로 명령어를 이슈하고 비순차 방식으로 명령어 수행을 끝내는 구조를 가지며[2][15][16], 2 개의 정수 기능 유닛(integer functional unit)과 5 개의 부동소수점/그래픽 기능 유닛(floating point/graphics functional unit), 1 개의 로드/스토어 유닛(Load/Store Unit, LSU) 및 1 개의 분기 유닛(branch unit)을 가지며 1 개의 8-윈도우(window) 정수 레지스터파일(register file)과 1 개의 부동소수점 레지스터파일을 가지고 있다. 정수 레지스터파일에는 7 개의 읽기 포트(read port)와 3 개의 쓰기 포트(write port)가 있다. 4 개의 읽기 포트와 2 개의 쓰기 포트는 정수 명령어를 위해 사용되며 2 개의 읽기 포트는 load/store 명령어의 데이터 주소 계산을 위해서 사용된다. 그리고 1 개의 읽기 포트가 정수 store 명령어를 위해 사용되고 1 개의 쓰기 포트는 정수 load 명령어에 사용된다. 부동소수점 레지스터파일에는 5 개의 읽기 포트와 3 개의 쓰기 포트가 있는데, 이 중에서 4 개의 읽기 포트와 2 개의 쓰기 포트는 부동소수점/그래픽 명령어를 위한 것이고 1 개의 읽기 포트와 1 개의 쓰기 포트는 각기 부동소수점 store 및 load 에 사용된다.

이슈 유닛은 프리페치 유닛에서 명령어를 받아서 정렬하여 저장한 후 순서대로(in order) 최대 4 개의 명령어를 단일 사이클에 실행 유닛에 이슈한다[2][17]. 그런데 각 실행 유닛의 수와 레지스터파일의 포트 수가 제한되고 데이터 의존성(data dependency) 때문에 실제로는 4 개의 명령어를 이슈할 수 없는 사이클이 생긴다. 이슈 유닛은 프리페치 유닛에서 입력되는 4 개의 명령어 중에서 유효한 첫번째 명령어를 명령어 버퍼의

첫째 입력 명령어가 되도록 명령어들의 순서를 바꾸어 출력하는 명령어 정렬기(instruction aligner)와 명령어들을 그룹으로 묶어서 최대 4 개의 명령어를 단일 사이클에 이슈하는 그룹화 로직(grouping logic)을 가지고 있다. 그리고 그 사이에 최대 12 개의 명령어를 저장할 수 있는 명령어 버퍼(instruction buffer)를 두어 명령어의 페치와 이슈를 분리(decoupling)시키고 명령어들을 정렬 및 병합(align and merge)한 후에 이슈할 수 있게 하여 명령어 이슈율(instruction issue rate)을 향상시키도록 하였다.

본 논문의 2 장은 전체 마이크로프로세서 및 이슈 유닛의 구조에 대하여 기술하였고, 3 장은 이슈 유닛을 구성하는 명령어 정렬기, 명령어 버퍼 및 그룹화 로직의 설계에 관해 자세히 다루었다. 4 장에서는 HDL(Hardware Description Language)을 사용한 이슈 유닛의 검증 및 모의실험(simulation)에 대해서 언급하였다. 마지막으로 4 장에서는 결론을 기술하였다.

제 2 장 마이크로프로세서 구조

2.1 절 4-way 수퍼스칼라 마이크로프로세서

본 논문의 마이크로프로세서는 SPARC-V9 을 기본 명령어 세트로 채택한 64 비트 4-way 수퍼스칼라 RISC(Reduced Instruction Set Computer) 마이크로프로세서로서, 그래픽 처리를 위해 영상 명령어 세트를 추가하였다[18][19]. 전체 마이크로프로세서는 프리페치 유닛(prefetch unit), 이슈 유닛(issue unit), 실행 유닛(execution unit), 레지스터파일(register file), 메모리 관리 유닛(Memory Management Unit, MMU), 외부 캐쉬(External Cache, E-Cache) 및 시스템 인터페이스(system interface)로 이루어졌다.

프리페치 유닛은 분기 예측(branch prediction) 기능을 지닌 명령어 캐쉬와 명령어 메모리 관리 유닛을 포함하는 것으로서 외부 캐쉬에서 명령어들을 받아 명령어 캐쉬에 저장하고 분기 예측을 사용하여 이슈할 명령어들을 신속하게 이슈 유닛에 공급하는 기능을 한다. 이슈 유닛은 프리페치 유닛에서 들어오는 명령어들을 받아서 임시로 저장하고서 멀티플렉서(multiplexer)를 이용하여 정렬하여 명령어 버퍼에 저장한 후에 명령어들을 실행 유닛들에 순서대로 이슈하는 기능을 한다. 실행 유닛에는 정수 실행 유닛(Integer Execution Unit, IEU), 부동소수점/그래픽 유닛(Floating point/Graphics Unit, FGU), 로드/스토어 유닛 및 분기 유닛이 있으며, 정수 실행 유닛은 2 개의 기능 유닛을 포함하며, 부동소수점/그래픽 유닛은 3 개의 부동소수점 기능 유닛(3-cycle latency floating-point adder and

multiplier, and nonpipelined divider/square root)과 그래픽 명령어 처리를 위한 2개의 그래픽 기능 유닛(four 8×16 bit multipliers and four 16-bit adders)을 포함한다. 실행 유닛은 비순차적 방식으로 명령어의 수행을 완료하는데 그 이유는 load/store 명령어 또는 부동소수점 나눗셈 및 제곱근 연산과 같이 지연 시간이 긴 동작의 수행과는 독립적으로 다른 명령어들을 수행할 수 있게 함으로써 전체적인 성능이 향상되기 때문이다. 레지스터파일에는 정수 레지스터파일과 부동소수점 레지스터파일이 있다. 정수 레지스터파일은 8개의 윈도우[20]로 이루어졌으며 7개의 읽기 포트와 3개의 쓰기 포트를 가지고 있다. 이 중에서 1개의 읽기 포트는 정수 store 명령어의 실행에만 사용되는 포트이며, 쓰기 포트 중에서 1개는 정수 load 명령어의 실행에만 사용된다. 부동소수점 레지스터파일은 32개의 64 비트 레지스터로 이루어졌고, 5개의 읽기 포트와 3개의 쓰기 포트를 가지고 있다. 부동소수점 레지스터파일의 경우도 1개의 읽기 포트는 부동소수점 store 명령어를 위해서만 사용되며, 1개의 쓰기 포트는 부동소수점 load 명령어의 실행에만 사용된다. 이와 같이 레지스터파일의 포트 수가 제한되었기 때문에 정수 명령어와 부동소수점 명령어는 단지 2개만을 단일 사이클에 이슈할 수 있으며 load/store 명령어는 1개의 명령어만 단일 사이클에 실행할 수 있다. 본 논문의 마이크로프로세서는 명령어와 데이터가 분리된 Harvard 아키텍쳐를 사용하기 때문에 메모리 관리 유닛에는 명령어 메모리 관리 유닛과 데이터 메모리 관리 유닛이 있다. 명령어 메모리 관리 유닛은 프리페치 유닛에서 명령어 캐쉬와 함께 명령어의 신속한 페치를 위해 사용되며, 데이터 메모리 관리 유닛은

데이터 캐쉬와 함께 로드/스토어 유닛의 일부로서 존재한다. 마이크로프로세서의 사양이 표 2-1 에 간략히 서술되어 있으며 전체 구조가 그림 2-1 에 나와 있다. 그림 2-1 의 외부 캐쉬는 칩 외부에 존재하도록 되어있다.

표 2-1. 4-way 수퍼스칼라 RISC 마이크로프로세서의 사양

명령어 세트	SPARC-V9 과 그래픽 지원 추가 명령어
최대 이슈 명령어 수	4 (4-way 수퍼스칼라 방식)
정수 레지스터파일	8 windows, 160 64-bit registers, 7 read ports, 3 write ports
부동소수점 레지스터파일	32 64-bit registers, 5 read ports, 3 write ports
정수 기능 유닛	2 개 (ALU0, ALU1 including multiplier and divider)
부동소수점/ 그래픽 기능 유닛	5 개 (floating point adder, multiplier, divider, four 8×16 bit multipliers, and four 16-bit adders)
I-Cache	16K 2-way set associative physically indexed, physically tagged , line size=32b
D-Cache	16K direct mapped, virtually indexed, physically tagged cache, write-through non-allocating, line size=32b
E-Cache	direct mapped, physically indexed, physically tagged cache, write-back allocating, size from 512K to 4M, line size=64b
데이터 버스	64-bit 내부 버스, 128-bit 외부 버스
instruction buffer	12 entries
load buffer	9 entries
store buffer	8 entries
I-MMU	44-bit virtual address, 41-bit physical address, page size = 8K, 64K, 512K or 4M, 64-entry fully associative TLB
D-MMU	44-bit virtual address, 41-bit physical address, page size = 8K, 64K, 512K or 4M, 64-entry fully associative TLB
파이프라인 단계 수	9 (F-A-G-E-C-N1-N2-N3-W)

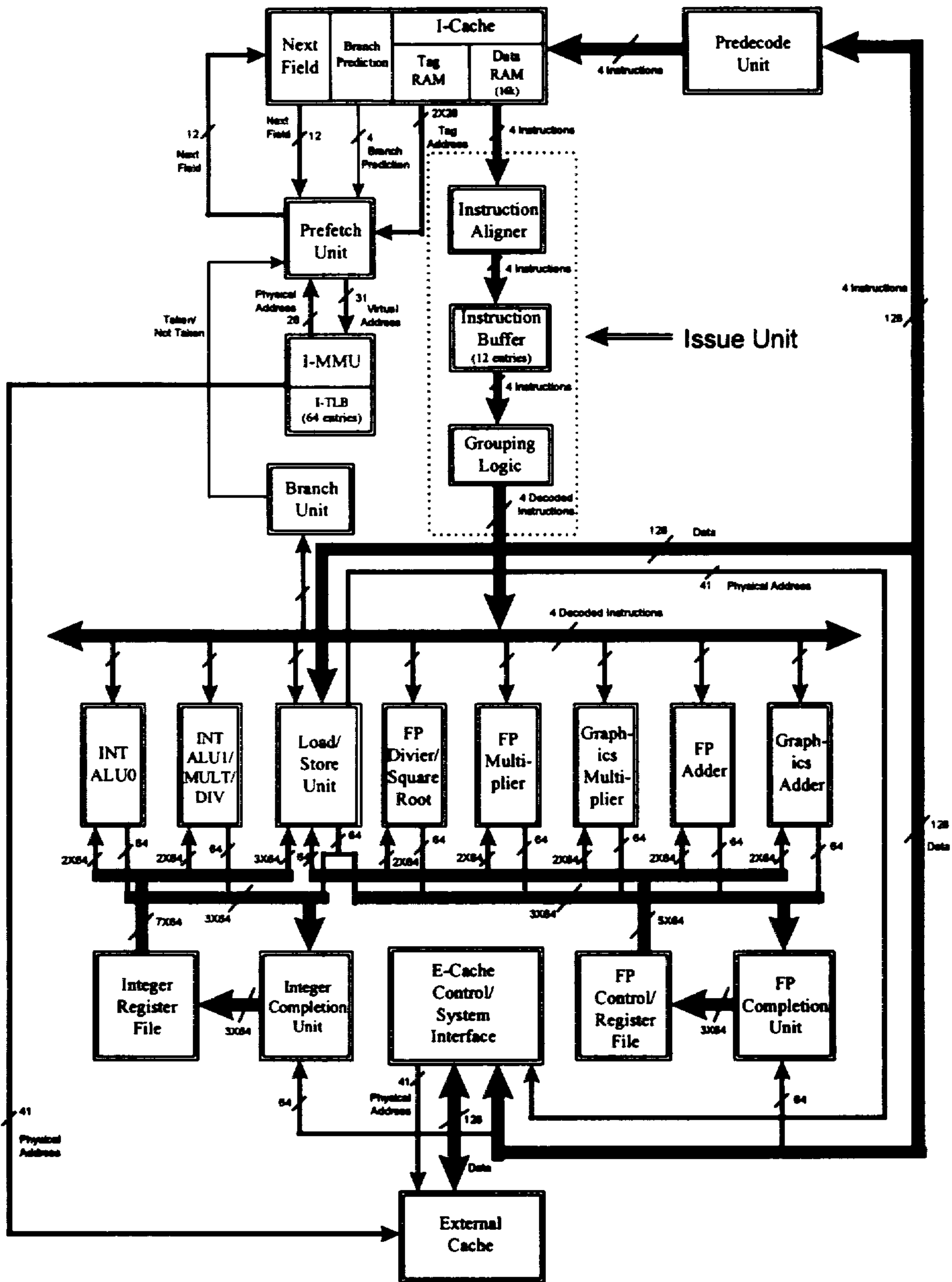


그림 2-1. 4-way 수퍼스칼라 마이크로프로세서의 전체 블럭다이어그램

마이크로프로세서의 정수 명령어 파이프라인은 6단계로 충분하지만 부동소수점 명령어와의 파이프라인 동기를 위해서 그림 2-2와 같이 3단계를 추가하여 9단 파이프라인을 사용하였으며 부동소수점 명령어의 파이프라인은 E-N2 단계 대신 R-X3 단계를 사용한다. 이와 같은 파이프라인의 사용으로 트랩(trap) 처리[21]가 간단해 졌으며 부동소수점 큐(floating-point queue)[7]에 대한 필요성이 없어졌다.

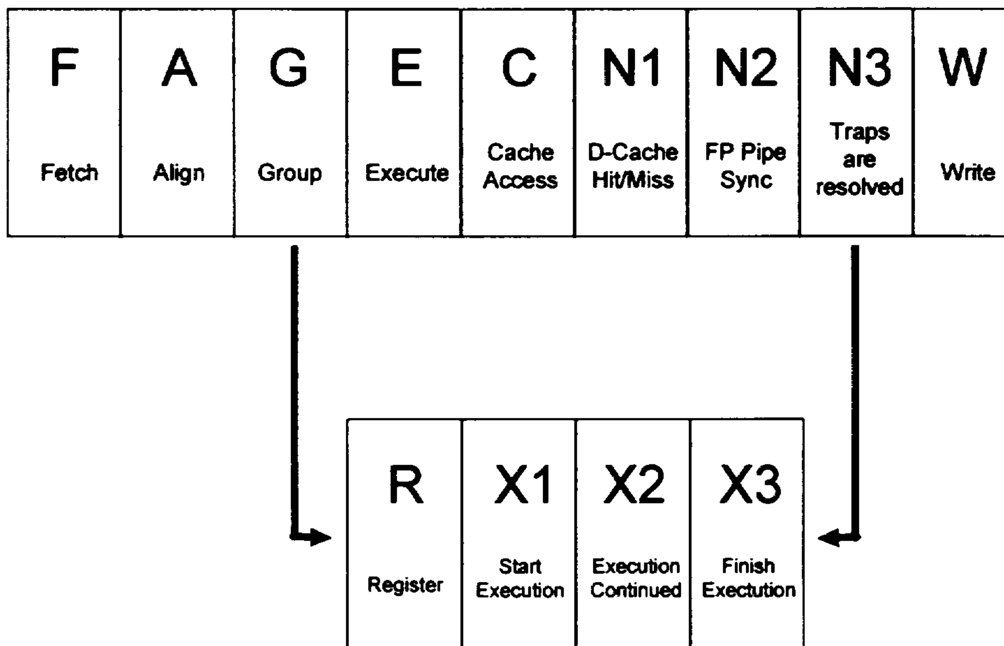


그림 2-2. 4-way 수퍼스칼라 마이크로프로세서의 9단 파이프라인

각 단계에서의 동작은 다음과 같다.

- F(Fetch) : 분기 예측을 사용하여 명령어 캐쉬로부터 최대 4 개의 명령어를 읽어온다.
- A(Align) : 페치한 명령어 중에서 첫번째로 유효한 명령어가 처음 명령어가 되도록 4 개의 명령어를 정렬하여 명령어 버퍼에 저장한다.
- G(Grouping) : 명령어 버퍼에 있는 명령어 중에서 가장 먼저 저장된 최대 4 개의 명령어를 해석하고 레지스터파일에서 피연산자(operand)를 읽은 후 순서대로 명령어를 이슈한다.
- E(Execute), R(Register) : 2 개의 ALU 를 사용하여 정수 명령어의 수행이 이루어지며, 이와 함께 load/store 명령어의 가상 주소(virtual address)가 계산된다. 부동소수점 명령어의 입력 데이터가 이 단계에서 읽혀진다.
- C(Cache Access), X1 : load/store 명령어의 가상 주소가 태그 램(tag RAM)에 보내지고 가상 주소를 이용해서 D-MMU 에서 실제 주소(physical address)가 계산되며 load 의 데이터가 읽혀진다. ALU 연산에 의한 조건 코드(condition code)가 결정되고 이에 따라 분기 예측이 제대로 되었는지가 결정된다. ALU 의 연산 결과는 단순히 파이프라인을 통해서 전달되고 부동소수점 명령어의 연산이 시작된다.
- N1, X2 : 데이터 캐쉬와 D-MMU 의 히트/미스(hit/miss)가 결정되고, 미스된 load 명령어는 로드 버퍼(load buffer)에 들어간다. store

명령어의 실제 주소가 스토어 버퍼(store buffer)에 입력되며, store 데이터가 사용 가능한 경우에 그 데이터도 스토어 버퍼에 들어간다. 부동소수점 명령어는 실행을 계속한다.

- N2, X3 : 대부분의 부동소수점 명령어들은 이 단계에서 계산이 끝난다.
- N3 : 트랩이 해결된다.
- W(Write) : 연산 결과가 레지스터파일에 저장되며, 이 단계를 지나면 명령어에 의해 변경된 상태는 되돌릴 수 없게 된다.

2.2 절 이슈 유닛

이슈 유닛은 단일 사이클에 최대 4 개의 유효한 명령어들을 프리페치 유닛으로부터 받아서 명령어 버퍼에 정렬하여 저장한다. 이렇게 저장된 명령어들은 단일 사이클에 최대 4 개까지 이슈된다. 그러나 기능 유닛과 레지스터파일의 포트의 수가 제한되었기 때문에 이슈 유닛은 단일 사이클에 최대 2 개의 정수 명령어, 최대 2 개의 부동소수점 명령어 및 1 개의 분기 명령어와 1 개의 load/store 명령어만을 이슈할 수 있다. 이슈 유닛은 명령어 이슈 외에도 파이프라인을 제어하고 2 종류의 오류(exception)를 검사한다.

이슈 유닛은 그림 2-3 과 같이 명령어 정렬기, 명령어 버퍼 및 그룹화 로직으로 구성되어 있다. 프리페치 유닛으로부터 입력된 최대 4 개의 명령어가 각 부분을 따라 흘러가고 각 명령어의 유효함(validness)을 나타내는 신호가 같이 전달된다. 그리고 그룹화 로직에서의 명령어 이슈에 필요한 신호들(del, pannul, align32)도 명령어 1 개에 1 개씩 같이 전달된다.

명령어 정렬기는 프리페치 유닛에서 최대 4 개의 유효한 명령어들을 받아서 이 중에서 첫번째 유효한 명령어가 명령어 버퍼의 첫째 입력 명령어가 되도록 정렬한 후 명령어 버퍼에 보내는 기능을 한다. 명령어 정렬기가 이와 같은 동작을 하는 것은 명령어 버퍼가 첫번째 명령어부터 차례로 저장하도록 되어있어서 유효한 명령어 앞에 있는 유효하지 않은 명령어는 제거할 필요가 있기 때문이다.

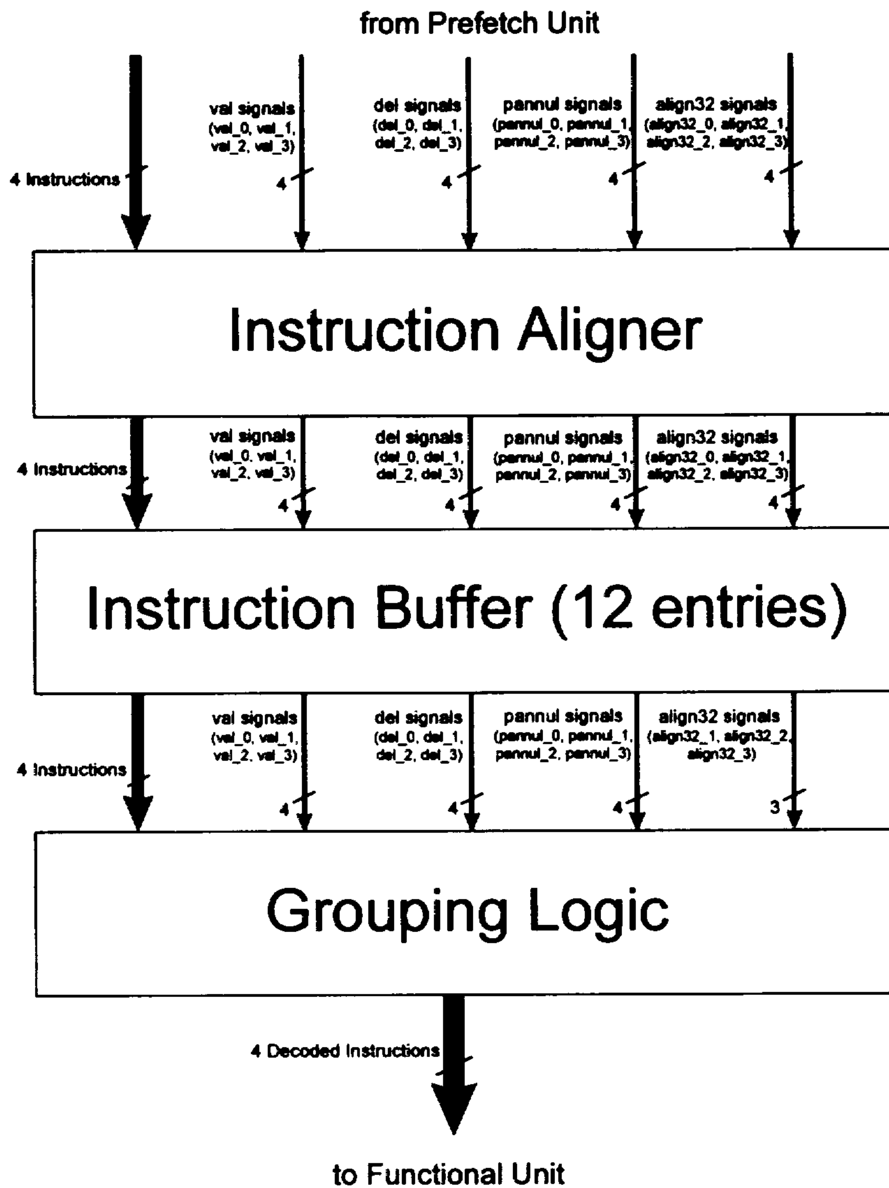


그림 2-3. 이슈 유닛의 블럭다이어그램

명령어 버퍼는 명령어 정렬기로부터 들어오는 명령어들을 받아서 12개의 명령어까지 저장할 수 있도록 되어있는 내부 저장 소자에 정렬하여 저장하고, 이 명령어들을 차례대로 그룹화 로직에 보내주어 명령어를 이슈하게 한다. 명령어를 이슈하기 전에 명령어 버퍼에 일시적으로 저장함으로써 명령어의 페치와 이슈를 분리할 수 있게 되어 명령어의 페치 또는 이슈 중에 어느 한 쪽이 중단되거나 느리게 되더라도 다른 한 쪽이 영향을 덜 받는다. 또한 명령어가 정렬된 상태로 그룹화 로직에 공급되므로 정렬되지 않은 상태로 명령어를 공급하는 경우보다 성능이 향상된다.

그룹화 로직은 명령어 버퍼로부터 오는 명령어들을 검사하여 이슈할 수 있는 명령어들을 그룹(group)으로 모아 단일 사이클에 최대 4개의 명령어를 이슈하는 역할을 한다. 그룹화 로직이 이슈할 수 있는 명령어를 결정하는 과정에 영향을 미치는 요소에는 사용 가능한 명령어의 수, 필요한 자원(resource, 레지스터파일과 실행 유닛)의 사용 가능성 및 데이터 의존성이 있다. 그룹화 로직에서 이슈된 명령어는 명령어 버퍼에서 지워지고 그 위치에는 새로운 명령어가 들어갈 수 있게 된다. 그룹화 로직은 이 외에도 실행 유닛에 연산 데이터를 제공하는 기능을 하여 레지스터파일에 레지스터 주소를 공급하며, 레지스터 주소 비교를 통해 데이터 의존성[22][23]을 검출하고 이에 따른 파이프라인 정지(pipeline stall)를 제어한다.

제 3 장 이슈 유닛의 설계

3.1 절 명령어 정렬기

3.1.1 절 명령어 정렬기의 동작

명령어 정렬기는 프리페치 유닛에서 들어오는 최대 4 개의 유효한 명령어들을 받아서 첫번째로 유효한 입력 명령어가 첫번째 출력 명령어가 되도록 명령어 순서를 바꾼 후에 이를 명령어 버퍼에 보내주는 역할을 한다. 명령어 버퍼는 명령어 정렬기에서 오는 첫번째 명령어부터 저장하므로 명령어 버퍼에 들어오는 4 개의 명령어는 모두 유효하지 않거나 또는 유효한 명령어들이 4 개의 명령어 중에서 가장 앞 부분에 있어야 한다. 명령어가 유효한지 또는 유효하지 않은지는 프리페치 유닛에서 각 명령어에 1 개씩 같이 보내주는 유효 비트(valid bit)들에 의해서 알 수 있다. 예를 들어 그림 3-1 과 같이 첫번째 명령어(inst_in0)가 유효하지 않고 나머지 세 명령어가 유효한 경우에 둘째부터 넷째까지의 입력 명령어(inst_in1, inst_in2, inst_in3)가 각기 첫째부터 셋째까지의 출력 명령어(inst_out0, inst_ou1, inst_out2)가 되어 출력되며 넷째 출력 명령어(inst_out3)는 의미 없는 명령어로서 넷째 입력 명령어(inst_in3)의 값을 받게 된다. 그리고 출력되는 명령어들의 유효성을 나타내기 위해 첫째부터 셋째까지의 출력 유효 비트들(val_out0, val_out1, val_out2)은 1 이 되고 넷째 출력 유효 비트(val_out3)는 0 이 되어 출력된다. 첫번째 명령어가 유효하지 않은 경우는 그림 3-2 와 같이 taken 될 것으로 예측된

분기 명령어의 목적 명령어(branch target instruction)가 캐쉬 line 의 끝에서 3 번째에 위치한 경우에 발생한다. 이 경우 목적 명령어를 페치하는 단계에서 단지 3 개의 명령어만을 페치하게 되어 첫번째 명령어는 유효하지 않은 명령어가 된다. 이 경우 외에도 2 개 또는 1 개의 명령어만 유효한 경우도 있다. 각 경우에 대한 입력 명령어들과 출력 명령어들 사이의 관계와 입력 유효 비트들과 출력 유효 비트들 사이의 관계가 표 3-1 에 설명되어 있다. 부수적인 입출력 신호들의 관계는 입출력 명령어들의 관계와 같다. 이와 같은 동작은 매 클럭 사이클마다 이루어지나 명령어 버퍼가 모두 채워진 경우와 같이 새로운 명령어들을 받아들일 수 없는 경우에는 그전의 명령어들을 그대로 유지한다.

표 3-1. 명령어 정렬기의 입출력 관계

val_in signals[1]	inst_out0	inst_out1	inst_out2	inst_out3	val_out signals[2]
1 1 1 1	inst_in0	inst_in1	inst_in2	inst_in3	1 1 1 1
1 1 1 0	inst_in0	inst_in1	inst_in2	inst_in3	1 1 1 0
0 1 1 1	inst_in1	inst_in2	inst_in3	inst_in3	1 1 1 0
1 1 0 0	inst_in0	inst_in1	inst_in2	inst_in3	1 1 0 0
0 1 1 0	inst_in1	inst_in2	inst_in3	inst_in3	1 1 0 0
0 0 1 1	inst_in2	inst_in3	inst_in3	inst_in3	1 1 0 0
1 0 0 0	inst_in0	inst_in1	inst_in2	inst_in3	1 0 0 0
0 1 0 0	inst_in1	inst_in2	inst_in3	inst_in3	1 0 0 0
0 0 1 0	inst_in2	inst_in3	inst_in3	inst_in3	1 0 0 0
0 0 0 1	inst_in3	inst_in3	inst_in3	inst_in3	1 0 0 0
0 0 0 0	inst_in3	inst_in3	inst_in3	inst_in3	0 0 0 0

1. val_in 신호의 값은 val_in0 val_in1 val_in2 val_in3 의 순서로 나열되어 있다
2. val_out 신호의 값은 val_out0 val_out1 val_out2 val_out3 의 순서로 나열되어 있다.

inst_in0 (invalid)	inst_in1 (valid)	inst_in2 (valid)	inst_in3 (valid)

Input Instructions

val_in0 = 0	val_in1 = 1	val_in2 = 1	val_in3 = 1

Input Valid Signals

inst_out0 =inst_in1 (valid)	inst_out1 =inst_in2 (valid)	inst_out2 =inst_in3 (valid)	inst_out3 =inst_in3 (invalid)

Output Instructions

val_out0 = 1	val_out1 = 1	val_out2 = 1	val_out3 = 0

Output Valid Signals

그림 3-1. 명령어 정렬기의 입출력 관계의 한 예

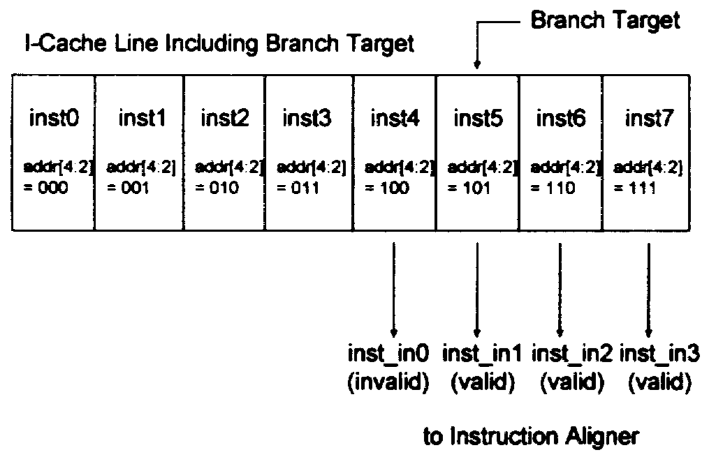


그림 3-2. 첫째 명령어가 무효한 경우

3.1.2 절 명령어 정렬기의 구조

명령어 정렬기에 입력되는 명령어와 관련 신호들은 우선 36 비트 레지스터에 저장되고, 그리고 나서 유효 비트를 제외한 신호들과 명령어들은 그림 3-3 과 같이 3 개의 멀티플렉서를 이용하여 정렬된다. 표 3-1 에서 알 수 있듯이 4 개의 입력 명령어 모두 inst_out0 가 될 수 있으므로 inst_out0 와 그 관련 신호들을 선택하는 멀티플렉서의 입력은 4 개이다. 같은 이유로 해서 inst_out1 을 선택하는 멀티플렉서의 입력은 3 개의며 inst_out2 에 대한 멀티플렉서는 입력이 2 개이다. inst_out3 는 항상 inst_in3 를 취하므로 멀티플렉서가 필요 없으며 inst_in3 가 저장된 것을 그대로 출력한다. 멀티플렉서들의 선택 신호들은 표 3-1 의 규칙에 따라 유효 비트들에 의해 만들며 그림 3-4 와 같이 만들어 진다. 그리고 그림 3-3 과 같이 멀티플렉서의 마지막 입력에 대한 선택 신호는 없는데, 그 이유는 모든 선택 신호들의 값이 0 일 때 마지막 입력이 출력으로 나오도록 되어있기 때문이다. 출력 유효 비트들도 표 3-1 에 따라 그림 3-5 와 같이 만들었다.

명령어 버퍼에 새로운 명령어를 저장할 충분한 공간이 없는 것을 알리는 신호가 stall_bc_b 이므로, stall_bc_b 의 값이 1 인 경우에 명령어 정렬기는 명령어 버퍼에 충분한 공간이 생길 때까지 명령어 버퍼에 저장되지 못한 명령어들을 유지하고 있어야 한다. 이러한 동작을 위해서 명령어 정렬기 레지스터들의 enable 입력으로 stall_bc_b 를 인버트(invert)한 신호를 사용하였다.

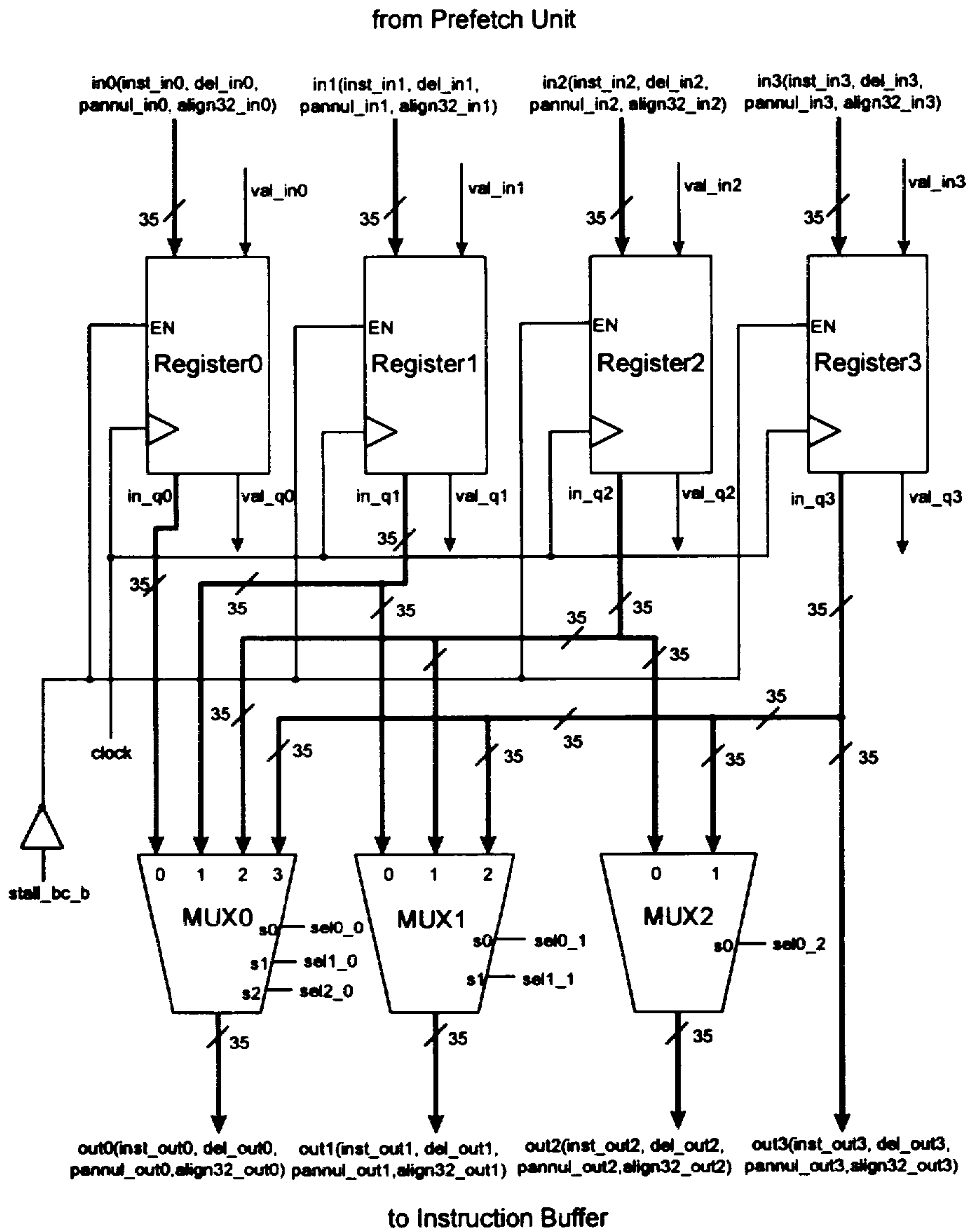


그림 3-3. 명령어 정렬기의 구조

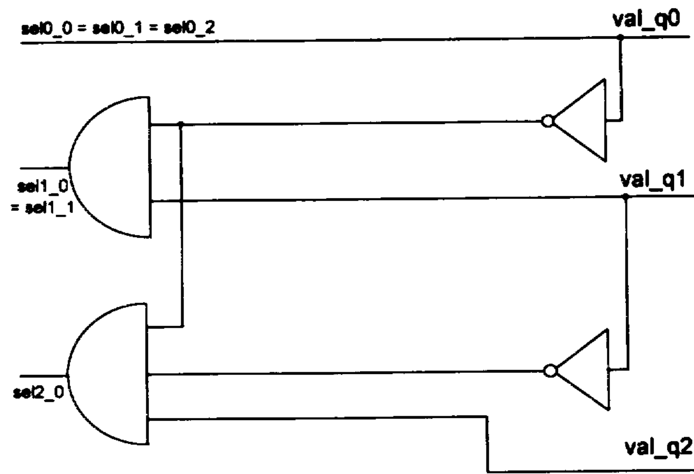


그림 3-4. 멀티플렉서 선택 신호 발생 로직

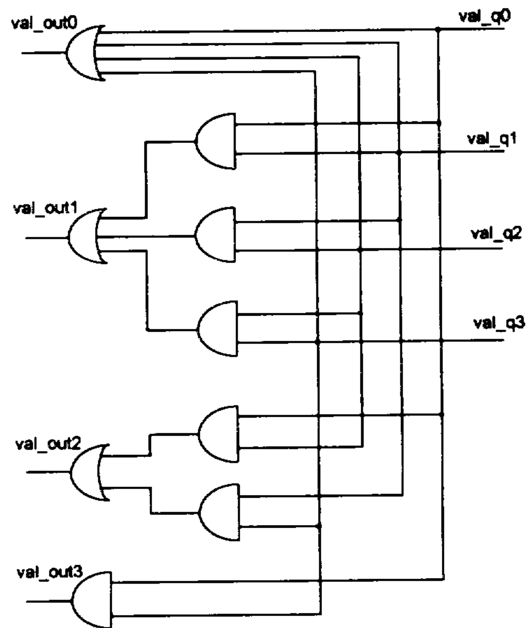


그림 3-5. 출력 유효 비트의 발생 로직

3.2 절 명령어 버퍼

3.2.1 절 명령어 버퍼의 동작

명령어 버퍼는 명령어 정렬기에서 입력되는 유효한 명령어들과 관련 신호들을 순서대로 정렬하여 내부 저장 소자(storage element)에 저장하고, 저장된 명령어들을 순서대로 그룹화 로직으로 보내주어 이슈하게 한다. 명령어 버퍼가 저장할 수 있는 명령어는 최대 12 개이다. 따라서 새로 입력되는 유효한 명령어들의 수가 비어있는 내부 저장 소자의 수보다 많은 경우에는 그 명령어들을 받아들일 수 없으며, 이러한 상황을 나타내는 신호를 명령어 정렬기에 보내어 명령어 버퍼의 명령어들이 이슈되어 새로운 명령어들을 받아들일 수 있을 때까지 기다리게 한다.

입력되는 명령어들이 들어갈 위치는 tail 이라는 포인터를 사용하여 나타내며, 그룹화 로직으로 내보낼 명령어들의 위치는 head 라는 포인터를 사용하여 알려준다. 새로 입력되는 유효한 명령어들은 tail 이 가리키는 위치부터 차례대로 저장되며 그 위치들의 유효 비트들은 1 이 된다. Tail 은 저장된 명령어 수만큼 증가하게 되어 다음 사이클에서 유효한 명령어들을 저장할 위치를 알려준다. Head 가 가리키는 위치부터 시작하여 4 개의 명령어들이 관련 신호들과 같이 그룹화 로직으로 출력되며 그 사이클에서 이슈가 결정된 명령어들은 그 명령어 위치의 유효 비트를 0 으로 만들어서 명령어 버퍼에서 없애고, head 는 이슈된 명령어 수만큼 증가하여 다음 사이클에서 그룹화 로직으로 내보낼 명령어 위치를 알려주게 된다.

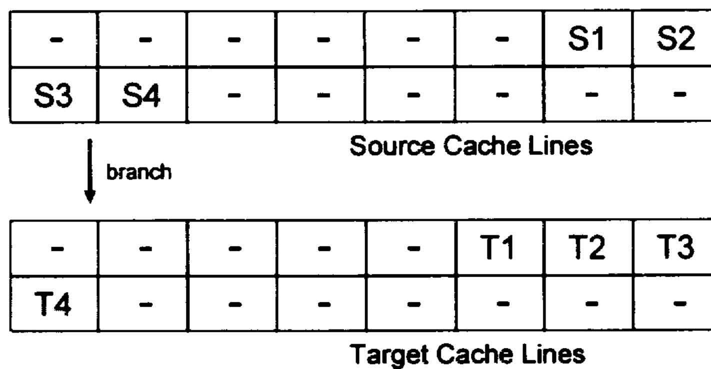
분기 예측 오류 또는 트랩이 발생하여서 파이프라인 단계에 있는 명령어들을 무효로 만들 필요가 있거나 시스템을 초기화하기 위해서는 명령어 버퍼의 저장 요소에 있는 모든 위치들의 명령어들을 무효화시켜야 한다. 이러한 경우에 명령어 버퍼는 모든 위치의 유효 비트들을 0으로 만들어서 명령어 버퍼를 비운다. 그리고 명령어 캐쉬에 미스가 발생하거나 I-TLB에 미스가 발생한 경우와 같이 유효한 명령어들이 들어올 수 없는 경우 명령어 버퍼는 명령어를 받아들이지 않는다.

페치된 명령어들을 명령어 버퍼에 일시적으로 저장한 후에 이슈하는 것은 명령어 이슈율을 높이기 위한 것이다. 프리페치 유닛에서 분기 예측에 의해 명령어들을 빠르게 공급하기는 하지만 4개의 명령어까지 이슈할 수 있는 이슈 유닛에서는 이것만으로는 충분한 명령어들을 받지 못한다. 따라서 명령어의 이슈와 명령어의 페치를 분리(decoupling)시킬 필요가 생긴 것이다. 이슈와 페치의 분리는 어느 한 쪽이 느리게 되거나 중단되더라도 다른 한 쪽은 영향을 받지 않게 하기 위한 것이다. 즉 명령어 이슈가 중단되더라도 명령어 버퍼에 명령어를 받아들일 충분한 공간이 있으면 명령어는 계속 페치된다. 이렇게 해서 후에 명령어 이슈가 다시 시작되고 명령어 페치가 느려지거나 중단되더라도 명령어 버퍼에 있는 명령어들을 사용하여 명령어 이슈를 계속 할 수 있다.

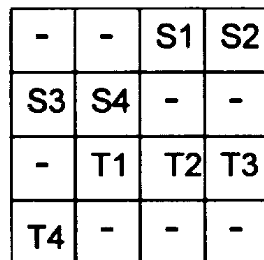
명령어 버퍼를 사용할 경우 또 하나의 장점이 있다. 중간에 분기가 있는 명령어 열(instruction stream)을 포함하는 캐쉬 라인들이 그림 3-6의 (a)와 같고 명령어 열에 포함된 명령어들은 서로 의존성도 없고 사용하는 기능 유닛이 충돌하지도 않는다고 할 때, 명령어 버퍼를 사용하지 않는

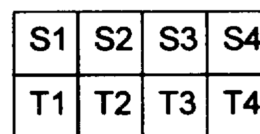
경우에는 그림 3-6 의 (b)와 같이 4 사이클 동안 이슈를 한다. 그러나 명령어 버퍼를 사용하는 경우에는 명령어들을 정렬(align) 및 병합(merge)하여[2] 이슈할 수 있도록 하기 때문에 그림 3-6 의 (c)와 같이 2 사이클 만에 이슈를 한다.

이와 같이 이슈 유닛이 명령어 버퍼를 사용함으로써 명령어의 페치 효율을 높여 전반적으로 명령어 이슈율을 높이도록 하였다.



(a) 중간에 분기가 있는 명령어 열을 포함하는 캐쉬 라인의 한 예





(b) (a) 명령어 열의 이슈 타이밍 (명령어 버퍼가 없을 때)

(c) (a) 명령어 열의 이슈 타이밍 (명령어 버퍼가 있을 때)

그림 3-6. 명령어 정렬과 명령어 이슈 타이밍

3.2.2 절 명령어 버퍼의 구조

명령어 버퍼는 그림 3-7과 같이 head 포인터와 tail 포인터를 조정하는 부분, 명령어 저장 어레이(instruction storage array) 및 명령어 저장 어레이의 입력과 출력을 결정 하는 부분으로 이루어져 있다.

그림 3-7에서 head는 현재 사이클에서 그룹화 로직으로 출력할 명령어들이 명령어 저장 어레이의 어디에 위치하는지를 알려주는 포인터이다. 명령어 저장 어레이의 구성 원소(entry)의 수가 12개이므로 head는 4 비트이다. 다음 사이클의 head는 이슈 또는 플러시(flush)된 명령어의 수(num_out)와 현재 사이클의 head로부터 구한다. 이슈된 명령어들의 수는 그룹화 로직으로부터 입력되는 issued (issued0, issued1, issued2, issued3)와 flushed0에 의해서 알 수 있다. 각 issued 신호는 그룹화 로직으로 보내진 명령어들의 이슈 여부를 알려주는 신호이며, flushed0는 첫번째 명령어가 취소되었다는 것을 알려 준다. 그림 3-8과 같이 head와 num_out을 더한 값(head_imm1)이 12(1100_2) 이상인 경우에 이를 보정하기 위해서 멀티플렉서(MUX1)를 사용하였다. Head_imm1이 12가 넘는 것은 head_imm1의 최상위 비트 2개를 입력으로 하는 AND 게이트의 출력(head_over)으로부터 알 수 있다. 그리고 명령어 버퍼를 초기화 하는 경우에(rst_buf=1) head가 0이 되도록 하기 위해서 또 하나의 멀티플렉서(MUX2)를 사용하였다.

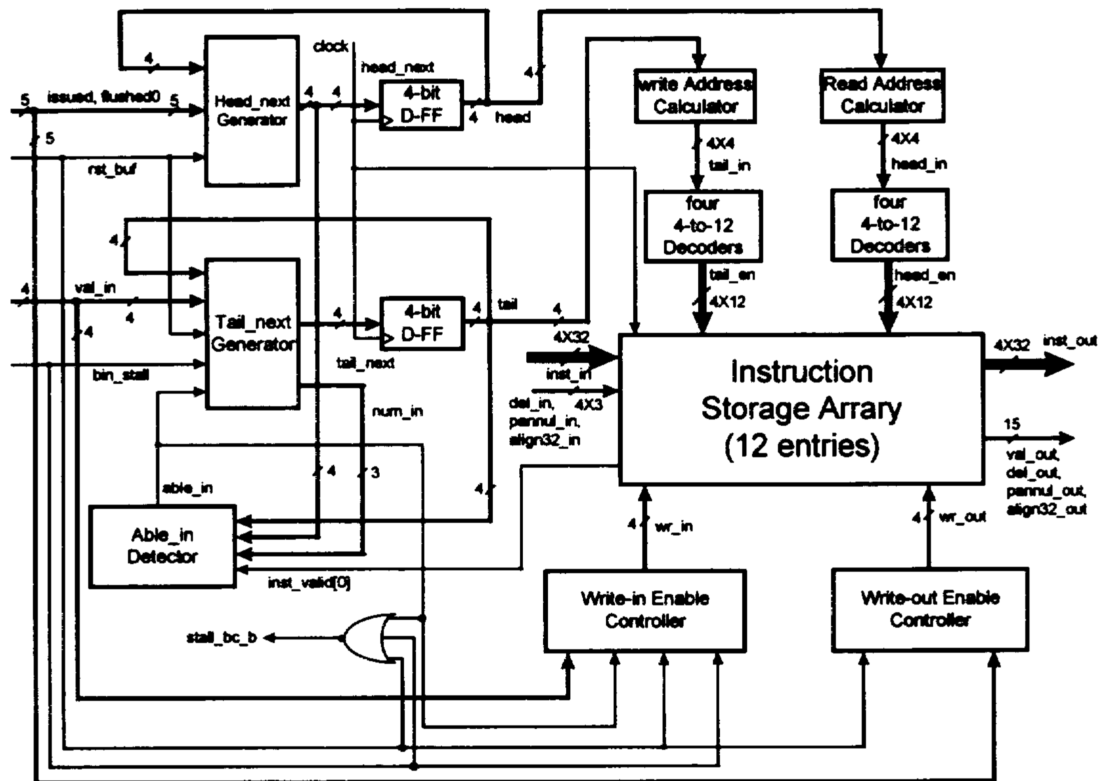


그림 3-7. 명령어 버퍼의 블럭다이어그램

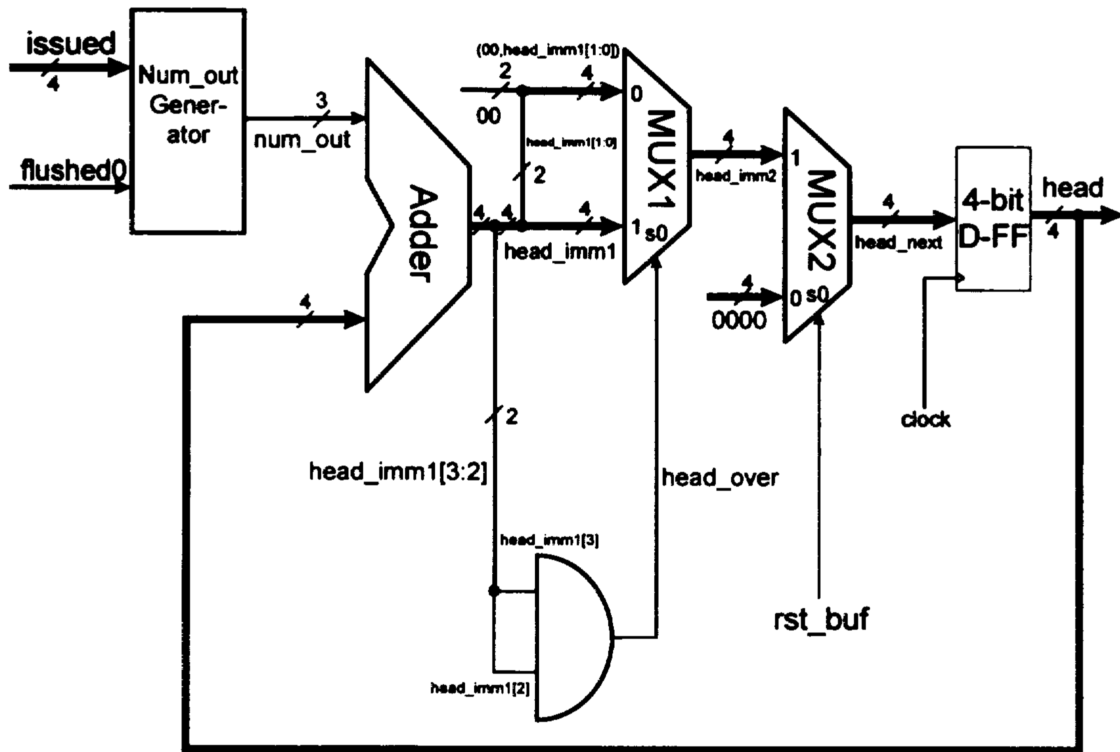


그림 3-8. head pointer 의 발생

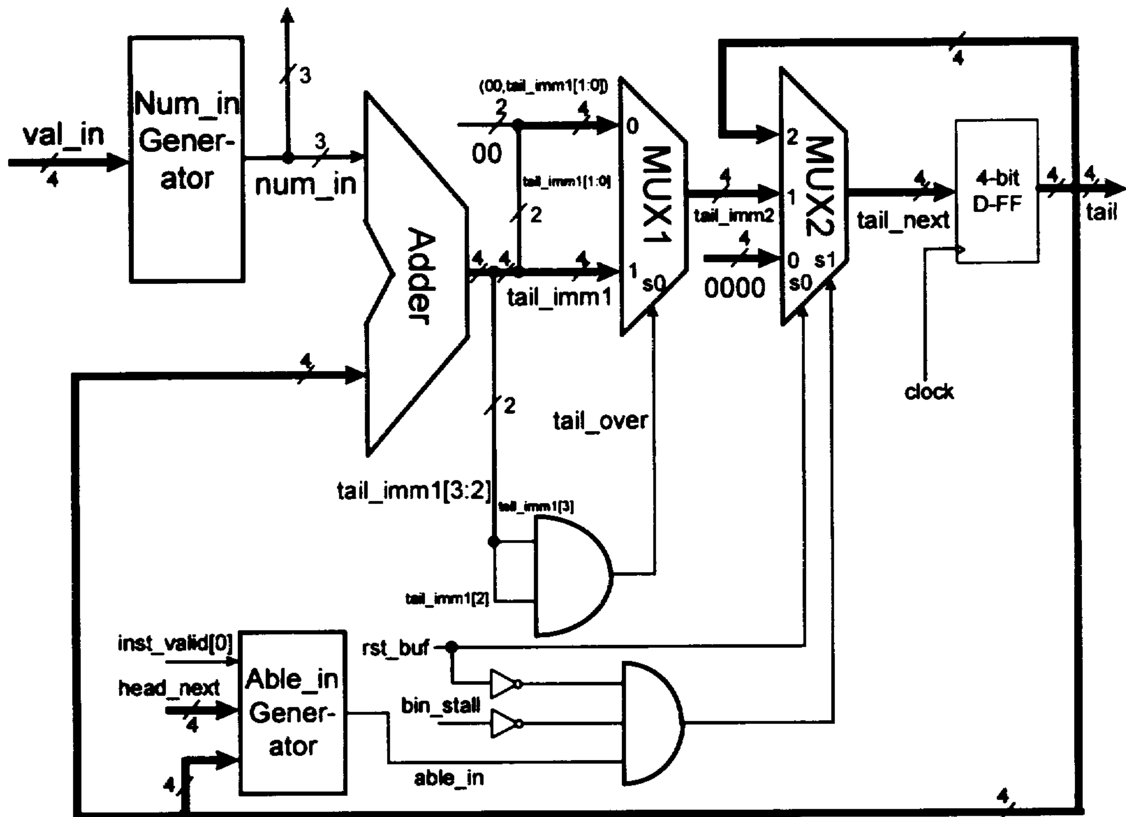


그림 3-9. tail pointer 의 발생

현재 사이클에 입력된 명령어들을 저장할 위치를 알려주는 신호가 tail 이다. Tail 은 head 와 같이 4 비트 포인터이지만, head 와는 달리 명령어 입력 명령어들의 유효성을 나타내기 위해서 명령어 정렬기로부터 입력되는 val_in(val_in0, val_in1, val_in2, val_in3) 신호에 의해서 다음 사이클의 tail 을 구한다. 그림 3-9 와 같이 유효한 명령어의 수(num_in)는 val_in 에 의해서 구하며 num_in 과 tail 을 더해서 다음 사이클의 tail 을 구한다. 그러나 더한 값(tail_imm1)이 12(1100_2)를 넘는 경우(tail_over=1)에 그 값을 조정하기 위해서 멀티플렉서(MUX1)를 사용하였다. 명령어 저장 어레이에 명령어를 저장할 공간이 충분하지 않을 경우에는 명령어를 받아들일 수 없다. 이러한 경우를 알려 주는 신호가 able_in 이며, 이것은 tail 과 다음 사이클의 head(head_next) 그리고 유효한 입력 명령어의 수(num_in)로부터 알아낸다. 그런데 tail 과 head_next 가 같은 경우 명령어 버퍼가 모두 비워 있는지 꽉 찬 경우인지를 구별할 필요가 있고 이것을 구별하기 위해서 명령어 저장 어레이의 첫번째 유효 비트(inst_valid[0])를 입력으로 받는다. Able_in 신호는 인버팅되어 외부로 출력됨으로써 현재 사이클에 입력된 명령어가 저장되지 않음을 알려준다. 명령어 버퍼를 초기화하거나, 명령어 버퍼로 명령어를 입력할 필요가 없는 경우(bin_stall=1)에도 tail 은 특수한 값을 갖게 되는데, 이를 위해서 두번째 멀티플렉서(MUX2)를 사용하였다.

명령어 저장 어레이에 명령어들을 쓰기 위해서는 저장할 부분에 대한 쓰기 제어 신호를 만들어 주어야 한다. Tail 부터 시작하는 4 개의 명령어의 주소(tail_in: tail0_in, tail1_in, tail2_in, tail3_in)를 계산하고 이것을 4-to-12 디코더(decoder)를 통해서 4 개의 주소에 대한 쓰기 제어 신호(write

enable, tail0_en: tail1_en, tail2_en, tail_3_en)를 만들어낸다. 그런데 4개의 주소를 디코딩 해서 얻은 쓰기 제어 신호가 1이더라도 입력 명령어를 그 위치에 쓸 수 있는 것은 아니다. 즉 입력 명령어 중에서 val_in 신호가 1이 아닌 명령어는 저장해서는 안된다. 그리고 명령어 버퍼에 명령어를 저장하지 않아야 하는 특별한 경우(bin_stall=1)와 명령어 저장 어레이에 여유 공간이 부족한 경우(able_in=0)에는 입력 명령어 모두 저장되지 않는다. 4개의 입력 제어 신호(wr_in)가 이러한 경우들을 제어한다. 이 제어 신호들에 의해 명령어 저장 레지스터는 입력되는 명령어들을 저장한다. 명령어들을 저장하면서 그 위치의 유효 비트들을 1로 만드는 것도 쓰기 제어 신호들에 의해서 알아낸다. 단 명령어 버퍼를 초기화 할 때 각 유효 비트는 0이 된다.

그룹화 로직으로 내보내야 할 명령어들을 선택하기 위해서 head로부터 시작하는 4개의 주소(head_in: head0_in, head1_in, head2_in, head3_in)를 계산하고 이것을 4-to-12 디코더로 디코딩하여 4개의 12 비트 읽기 제어 신호(read enable, head_en: head0_en, head1_en, head2_en, head3_en)를 발생시킨다. 읽기 제어 신호는 명령어 저장 어레이에서 출력 명령어를 선택하기 위한 멀티플렉서의 선택 신호(select signal)들로 사용된다. 출력되어 나가는 명령어 중에서 이슈된 명령어들의 유효 비트들은 0으로 만들어 명령어 저장 어레이에서 그 위치를 다른 명령어가 사용할 수 있게 한다. 읽기 제어 신호와 이슈된 명령어를 알려주는 신호(issued)가 명령어 버퍼에서 비워야 할 명령어의 위치를 알려주는 역할을 한다. 단 명령어 버퍼를 초기화할 때는 모든 유효 비트가 0이 된다. 그리고 어떤 위치에

있는 명령어가 이슈되었어도 다음 사이클에 같은 위치에 새로운 명령어가 저장되는 경우 그 위치의 유효 비트는 1 인 상태로 남게 된다.

3.3 절 그룹화 로직

3.3.1 절 그룹화 로직의 동작

그룹화 로직은 파이프라인 단계 중에서 G(group) 단계에 해당하는 부분이다. 그룹화 로직은 단일 사이클에 명령어 버퍼로부터 입력되는 최대 4 개의 유효한 명령어들을 각 실행 유닛에 이슈하고, 인터락(interlock)을 검출하기 위해서 레지스터 주소들을 비교하며, 필요한 경우 파이프라인을 정지(stall)시킨다.

그룹화 로직은 단일 사이클에 최대 4 개의 명령을 이슈할 수 있으나 다음과 같은 제한으로 명령어 이슈율이 낮아진다.

- 명령어 버퍼에 충분한 명령어가 없을 때 : 명령어 버퍼에 명령어가 없거나 3 개 이하의 명령어가 존재하는 경우에는 충분한 이슈가 일어날 수 없다.
- true(read-after-write) dependency[2] : 그룹화 로직에 있는 명령어가 아직 실행이 끝나지 않은 이전 명령어의 결과 데이터를 오퍼랜드로 사용하는 경우 그 명령어는 이슈될 수 없다.
- output(write-after-write) dependency[2] : 그룹화 로직의 명령어를 현재 사이클에 이슈하면 같은 레지스터에 결과를 쓰는 이전 명령어보다 먼저 데이터를 쓰게 되거나 같은 사이클에 쓰게 되는 경우 그 명령어를 현재 사이클에 이슈할 수 없다.
- anti-(write-after-read) dependency[2] : 그룹화 로직은 프로그램 순서대로

명령어들을 받으며 그 순서대로 이슈하므로 고려되지 않는다.

- 명령어의 종류 : 정수 명령어는 단일 사이클에 최대 2개만을 이슈할 수 있으며, 부동 소수점 명령어도 최대 2개의 이슈가 가능하다. load/store 명령어와 분기 명령어는 단일 사이클에 각기 1개의 명령어만을 이슈할 수 있다.
- 기능 유닛과 레지스터파일 포트의 제한 : 그룹화 로직에 있는 2개의 명령어가 같은 기능 유닛이나 레지스터파일의 포트를 사용해야 하는 경우 뒤의 명령어는 이슈되지 않는다. 예를 들어 단일 사이클에 2개의 정수 명령어가 이슈될 수 있지만, 2개의 명령어가 같은 기능 유닛을 사용해야 하는 경우 뒤의 정수 명령어는 이슈되지 못한다.

그룹화 로직은 이러한 제한과 3.3.2절에서 설명되어 있는 세부적인 그룹화 규칙에 의해서 명령어들을 각 실행 유닛에 이슈하며, 명령어 자체로부터 검출할 수 있는 오류를 검사한다.

3.3.2 절 그룹화 규칙

그룹화 로직은 3.3.1절에서 설명된 제한들을 고려해서 각 명령어들에 대한 세부적인 그룹화 규칙을 만들고 이에 따라 명령어들을 이슈한다. 그룹화 규칙[10]이란 그룹화 로직이 명령어들을 그룹으로 만들어 이슈할 때 기준으로 사용하는 이슈 규칙을 말한다. 명령어들은 사용하는 실행 유닛 및 이슈 특성에 따라서 단일 그룹 명령어, 정수 명령어, 분기 명령어, load/store 명령어 및 부동소수점/그래픽 명령어로 나뉜다. 같은 종류의

명령어들은 같은 실행 유닛과 레지스터파일을 사용하기 때문에 상호 영향을 많이 준다. 따라서 같은 종류의 명령어들 사이에는 많은 제한 규칙이 존재한다.

(1) 단일 그룹 명령어

단일 사이클에 그 명령어 하나만을 이슈할 수 있는 명령어들이 단일 그룹 명령어에 포함된다. 이 명령어들은 동작이 복잡하고 실행의 결과가 다른 명령어들의 실행에 많은 영향을 주는 명령어 또는 다른 명령어의 실행에 의해 많은 영향을 받는 명령어들이다. 예를 들어 상태 레지스터의 값을 변경하는 명령어의 실행은 다음에 오는 거의 모든 명령어의 실행에 영향을 주기 때문에 다른 명령어들과 같이 실행될 수 없다.

단일 그룹 명령어는 다음의 명령어들을 포함한다.

- LDD{A}, STD{A}, block load instructions, ADDC{cc}, SUBC{cc}, {F}MOVcc, {F}MOVr, SAVE, RESOTRE, {U,S}MUL{cc}, MULX, MULScc, {U,S}DIV{cc}, {U,S}DIVX, LDSTUB{A}, SWAP{A}, CAS{X}A, LD{X}FSR, ST{X}FSR, SAVED, RESOTORED, FLUSH{W}, ALIGNADDR, RETURN, DONE, RETRY, WR{PR,SR}, RD{PR,SR}, Tcc, SHUTDOWN, DCTI(Delayed Control Transfer Instruction) 쌍의 둘째 분기 명령어.

(2) 정수 명령어

정수 실행 유닛 또는 정수 레지스터파일을 사용하는 명령어들이 이

종류에 속한다. 정수 실행 유닛은 2개의 개별적인 기능 유닛을 가지고 있으며, 각 기능 유닛은 정수 레지스터파일에서 2개의 읽기 포트와 1개의 쓰기 포트가 고정적으로 할당되어 있기 때문에 사용하는 기능 유닛 또는 포트에 따라 표 3-2와 같이 정수 명령어를 세분할 수 있다.

표 3-2. 정수 명령어의 분류

종류	명령어
불특정 유닛 명령어	ADD, SUB, AND, ANDN, OR, ORN, XOR, XNOR, SETHI
유닛 0 명령어	SLL{X}, SRL{X}, SRA{X}
유닛 1 명령어	ADDC{cc}, ADDcc, SUBC{cc}, SUBcc, ANDcc, ANDNcc, ORcc, ORNcc, XORcc, XNORcc, TADDcc{TV}, TSUBcc{TV}, {U,S}MUL{cc}, MULX, MULScc, {U,S}DIV{cc}, {U,S}DIVX, SAVE, RESTORE, MOVcc, {F}MOVr, ALIGNADDR, RETURN, WR{PR,SR}, RD{PR,SR}, Tcc, EDGE, ARRAY, CALL, JMPL, BPr, PST, FCMP{LE,NE,GT,EQ}{16,32}

※ 정수 명령어 중에는 정수 실행 유닛을 사용하지는 않지만 CALL, JMPL, PST 등과 같이 정수 레지스터파일을 사용하기 때문에 정수 명령어에 속하는 것들도 있다.

정수 실행 유닛은 2개의 기능 유닛을 포함하므로 2개의 정수 명령어를 단일 사이클에 이슈할 수 있으며 다음과 같은 규칙에 따라 이슈한다.

■ 다중 사이클 명령어 : 실행에 여러 사이클이 걸리는 명령어를 다중 사이클 명령어라 하며, 이러한 명령어는 E 단계에 여러 사이클 동안 머무른다. 정수 명령어에 대해서는 순차적 방식의 수행 완료(completion)만이 허용되므로 다중 사이클 명령어 다음에 오는 명령어들은 이슈되지 못하고 계속 G 단계에 머무르게 된다. 표 3-3에 다중 사이클 명령어의 종류와 이슈 정지 사이클 수가 나와 있다.

표 3-3. 다중 사이클 명령어와 bubble 수

다중 사이클 명령어	bubble 수
MULScc (Multiply Step)	1
UMUL{cc} (Unsigned 32-bit Integer Multiply[1]	from 4 to 19
SMUL{cc} (Signed 32-bit Integer Multiply)[1]	from 4 to 18
MULX (64-bit Integer Multiply)[1]	from 4 to 34
UDIV{cc} (Unsigned 32-bit Integer Divide)	37
SDIV{cc} (Signed 32-bit Integer Divide)	36
{U,S}DIVX{64-bit Integer Divide}	68
WRPR (Write Privileged Register)	4
WRSR (Write State Register)	4

1. 곱셈 명령어들({U,S}MUL{cc}, MULX)은 early result 방식을 사용하므로 입력 오퍼랜드에 따라서 실행 사이클 수가 2 사이클 단위로 변한다[10].

- RD{PR,SR} : 이 명령어들은 그룹화 로직의 첫째 명령어로 입력되고 4 사이클 후에 이슈될 수 있다. 이 명령어들은 거의 대부분의 명령어들이 그 실행 결과에 따라 변경할 수 있는 값을 읽기 때문에 앞의 명령어들의 실행으로 인한 상태 레지스터의 값이 나온 후에 이슈되어야 하기 때문이다.
- true dependency : 이전 명령어의 결과를 사용하는 정수 명령어는 그 명령어의 결과가 나올 때까지 이슈되지 못한다. 이 경우 결과는 데이터 바이패스(data bypass)에 의해 전달되며 그 예가 그림 3-10 에 나와 있다.
- output dependency : 서로 같은 레지스터에 결과를 쓰는 2 개의 정수 명령어는 같은 사이클에 이슈되지 못한다. 그림 3-11 에 그 예가 나와 있다.
- 기능 유닛의 충돌 : 같은 기능 유닛을 사용하는 2 개의 명령어는 같은 사이클에 이슈될 수 없다. 즉 2 개의 유닛 0 명령어 또는 2 개의 유닛 1 명령어는 같은 사이클에 이슈될 수 없다. 불특정 유닛 명령어 다음에 유닛 0 명령어가 오는 경우, 불특정 유닛 명령어가 정수 기능 유닛 0 를 사용하기 때문에 뒤에 오는 유닛 0 명령어는 같이 이슈될 수 없다. 그러나 불특정 유닛 명령어 다음에 유닛 1 명령어가 오는 경우에는 같이 이슈될 수 있다. 그림 3-12 에 예가 나와 있다.

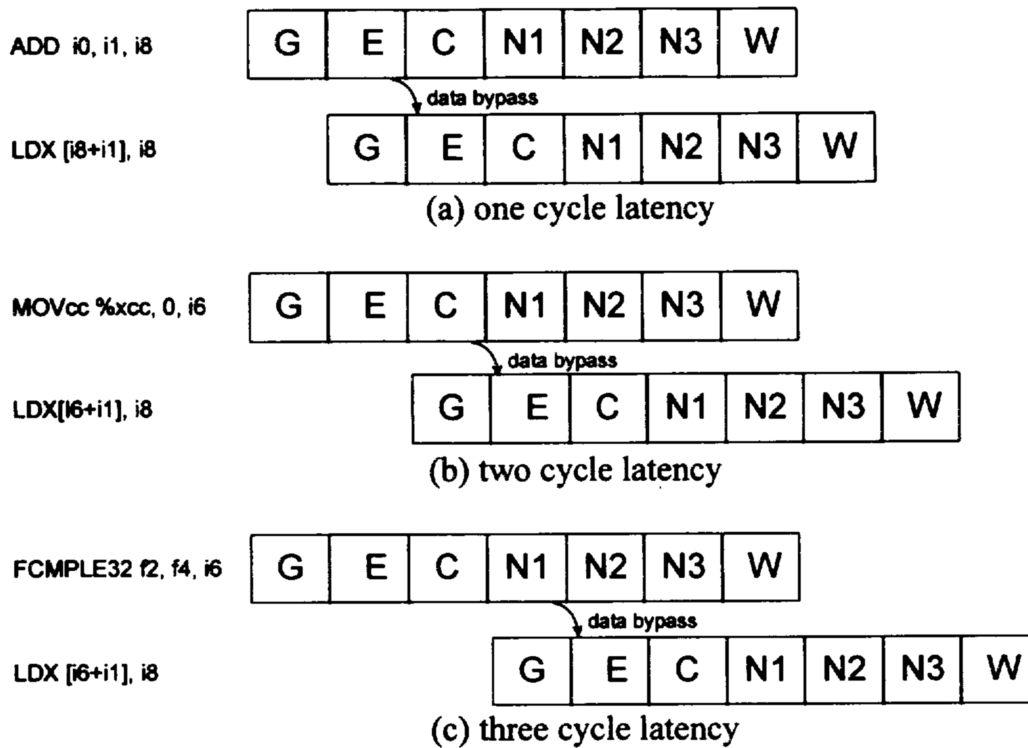


그림 3-10. 정수 명령어의 true dependency

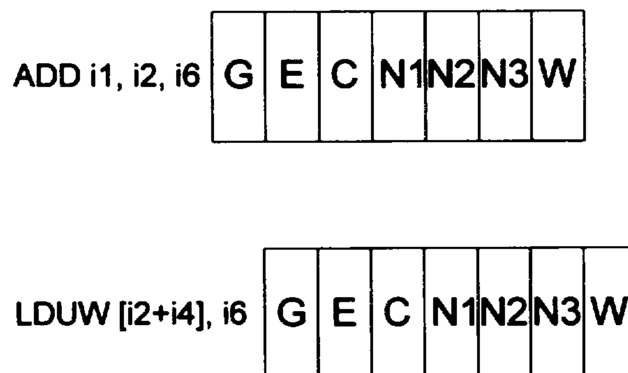


그림 3-11. 정수 명령어의 output dependency

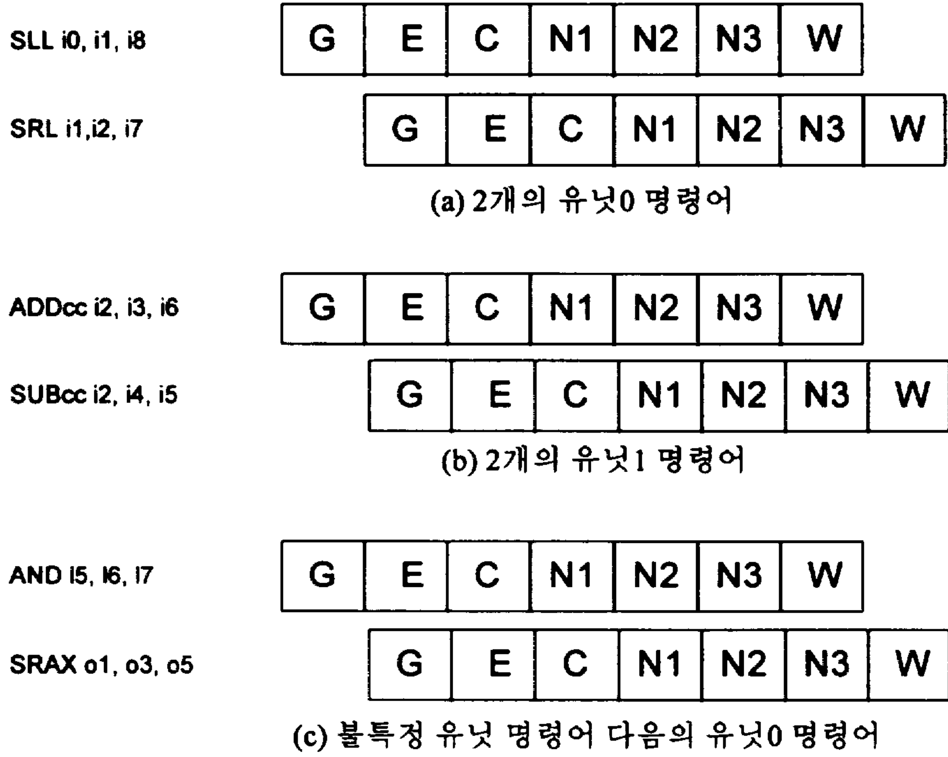


그림 3-12. 정수 기능 유닛의 충돌

■ 넷째 명령어 : 그룹화 로직으로 들어 오는 4개의 명령어 중에서 넷째 명령어로 입력되는 명령어가 정수 명령어인 경우 그 명령어는 이슈되지 못한다. 이것은 설계의 복잡성(complexity)과 하드웨어 비용을 줄이기 위한 결과이다. 다시 말해서 4개의 명령어 중에서 정수 실행 유닛을 사용할 수 있는 명령어의 수는 정수 레지스터파일의 포트나 정수 실행 유닛에 대한 명령어 할당을 조정하는 로직의 복잡성에 많은 영향을 준다[2]. 성능을 고려해서 앞의 3개 명령어까지만 정수 실행 유닛을 사용할 수 있게 하였다.

(3) 분기 명령어

명령어의 흐름을 바꾸고 분기 유닛을 사용하는 명령어가 이 종류에 속하며 다음과 같은 명령어들이 있다.

- CALL, JMPL, Bicc, BPcc, FB{P}fcc, BPr, RETURN, DONE, RETRY

분기 유닛은 1 개만이 존재하므로 단일 사이클에 이슈할 수 있는 분기 명령어의 수는 1 개이며 다음과 같은 규칙에 따라 이슈된다.

- 분기 명령어 이전이나 다음에 오는 명령어들은 분기 명령어와 같이 이슈될 수 있다. 그림 3-13 에 그 예가 나와 있다.
- 그림 3-13 에 나와 있듯이 정수 조건 코드를 변경하는 정수 명령어 다음에 그 조건을 사용하는 분기가 올 때 같이 이슈될 수 있다. 조건 분기가 결과를 출력하는 단계가 C 단계이기 때문에 앞의 정수 명령어를 기다릴 필요가 없다.
- 무효화 분기(annulling branch) : 무효화 분기가 not taken 으로 예상되어도 실제로는 taken 될 수 있기 때문에 지연 명령어(delay instruction)는 페치되고 이슈되어 데이터 의존성에 영향을 미친다.
- 무효화 될 것으로 예상되는 load 명령어는 데이터 의존성에 영향을 미치지 않는다. 즉 캐쉬 미스가 발생하는 경우와 같이 load 의 지연 시간이 매우 길어지는 경우가 생기므로 무효화 될 것으로 예상되는 load 명령어는 데이터 의존성 검사에서 고려하지 않는다.

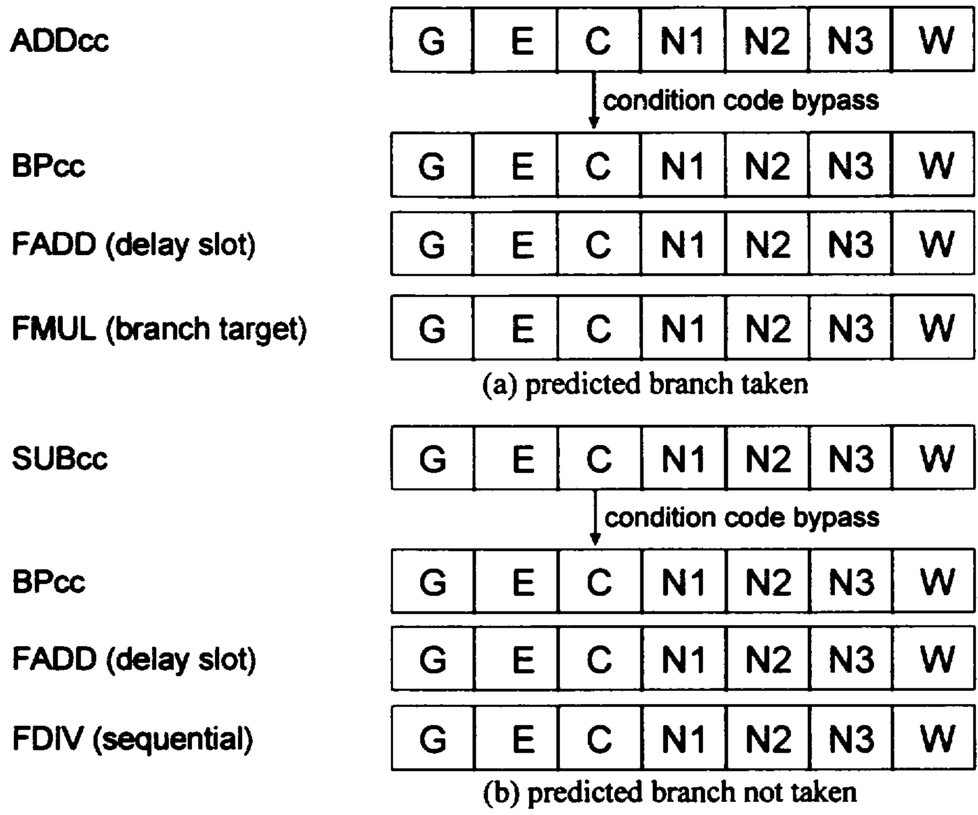


그림 3-13. 분기 명령어와 주변 명령어들의 이슈

(4) Load/Store 명령어

load 나 store 처럼 메모리와 캐쉬에서 데이터를 읽어오거나 쓰는 명령어들이 이 종류에 속하며 다음과 같은 것들이 있다.

- LD{SB,SH,SW,UB,UH,UW,X}{A}, LD{D}F{A}, ST{B,H,W,X}A, ST{D}F{A}, JMPL, FLUSH, MEMBAR, STBAR, PREFETCH{A}

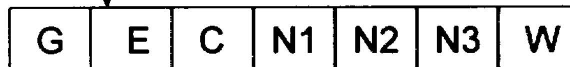
로드/스토어 유닛은 1 개만 존재하기 때문에 이 종류의 명령어는 단일 사이클에 1 개만을 이슈할 수 있으며 다음과 같은 규칙에 의해 이슈된다.

- 다중 사이클 명령어 : LDD{A}, STD{A}, LDSTUB{A}, SWAP{A}는 1 사이클 동안, CAS{X}A 는 2 사이클 동안 다음 명령어가 이슈되지 못하도록 한다. 이 명령어들은 캐쉬 또는 메모리에 대한 동작이 복합적이어서 1 사이클에 그 실행이 끝나지 않고 E 단계에 여러 사이클 동안 머무르기 때문이다.
- load 의존성 : load 는 캐쉬 히트의 경우 데이터를 C 단계에서 보내주기 때문에 load 의 결과를 사용하는 명령어는 그림 3-14 와 같이 load 의 G 와 E 단계에서 이슈되지 못한다. 설계상의 편의를 위해 단정도 부동소수점 load 명령어는 그 단정도 레지스터를 포함하는 배정도 레지스터에 대한 데이터 의존성에 영향을 준다. 그리고 load 는 캐쉬 미스가 발생하면 로드 버퍼에 들어가고 out-of-order 방식으로 결과를 레지스터파일에 쓰기 때문에 output dependency 가 있는 명령어는 true dependency 가 있는 명령어와 같이 취급된다.

LDX [i1+i3], i5
(not enqueued)

load data bypass

AND i5, i6, i7



(a) true dependency

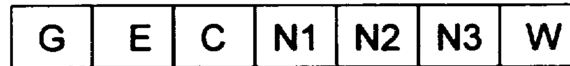
LDF [o2+o3], f6
(not enqueued)

FADD f7, f7, f8

(b) 같은 배정도 레지스터에 속하는 다른 단정도 레지스터에 대한 데이터 의존성의 예

LDUH [i3+i4], i6
(not enqueued)

ADD i2, i1, i6



(c) Output Dependency

그림 3-14. load 명령어에 대한 데이터 의존성

- store 의존성 : store 명령어에 의해 메모리에 쓰여질 데이터가 이전의 명령어에 의해 아직 계산되고 있어도 store는 이슈될 수 있다. store는 이슈된 후 곧바로 메모리를 접근하는 것이 아니라 우선 스토어 버퍼에 들어가기 때문에 메모리에 쓸 데이터가 준비되지 않았어도 후에 그 결과를 스토어 버퍼로 보낸 후에 메모리에 접근하면 된다.
- 캐쉬 미스로 인한 파이프라인 정지 : 캐쉬 히트/미스가 결정되는 것은 load 명령어의 N1 단계이다. 따라서 load의 C 단계에서 load 결과를 사용하는 명령어가 그룹화 로직에서 이슈되었는데, load의 N1 단계에서 캐쉬 미스가 결정된 경우에는 load의 결과를 사용하는 명령어는 잘못된 데이터를 받은 것이기 때문에 E 단계와 그 이전 단계의 모든 명령어들은 파이프라인에서 정지되며, 후에 데이터가 들어오면 다시 동작을 시작한다.
- 로드 버퍼의 제한 : 로드 버퍼가 받아들일 수 있는 명령어의 수는 9개로 제한 되었기 때문에, 이미 9개의 명령어가 실행되고 있을 때에 로드 버퍼를 사용하는 명령어는 이슈되지 못한다.
- 스토어 버퍼의 제한 : 스토어 버퍼가 받아들일 수 있는 명령어의 수는 8개로 제한되었기 때문에, 이미 8개의 명령어가 실행되고 있을 때에 스토어 버퍼를 사용하는 명령어는 이슈되지 못한다.
- 넷째 명령어 : 정수 명령어와 같은 이유로 그룹화 로직에 입력된 4개의 명령어 중에서 마지막 명령어가 load/store 명령어인 경우 그 명령어를 이슈할 수 없다.

(5) 부동소수점/그래픽 명령어

부동소수점 레지스터파일을 사용하거나 부동소수점/그래픽 유닛을 사용하는 명령어가 이 종류에 속한다. 부동소수점/그래픽 유닛 속에는 5 개의 기능 유닛이 있지만 할당된 레지스터파일 포트 수(4 개의 읽기 포트와 2 개의 쓰기 포트)는 2 개의 명령어를 지원하는 것에 지나지 않기 때문에 단일 사이클에 이슈할 수 있는 이 종류의 명령어는 2 개이다. 5 개의 기능 유닛은 2 개의 종류(class)로 나뉘고 각기 2 개의 읽기 포트와 1 개의 쓰기 포트를 사용한다. 부동소수점/그래픽 명령어는 2 개의 기능 유닛 종류 중에서 어느 하나에서만 실행할 수 있도록 되어있으며, 사용하는 기능 유닛 종류에 따라 표 3-4 와 같이 나뉜다.

표 3-4. 부동소수점/그래픽 명령어의 분류

종류	명령어
A class	F{i,x}TO{s,d}, F{s,d}TO{d,s}, F{s,d}TO{i,x}, FABS{s,d}, FADD{s,d}, FALIGNDATA, FAND{s}, FANDNOT1{s}, FANDNOT2{s}, FCMP{E}{s,d}, FEXPAND, FMOVr{s,d}, FMOV{s,d}cc, FNAND{s}, FNEG{s,d}, FNOR{s}, FNOT1{s}, FNOT2{s}, FONE{s}, FOR{s}, FORNOT1{s}, FORNOT2{s}, FPADD{16,32}s, FPMERGE, FPSUB{16,32}{s}, FSRC1{s}, FSRC2{s}, FSUB{s,d}, FXNOR{s}, FXOR{s}, FZERO{s}
M class	FCMP{LE,NE,GT,EQ}{16,32}, FDIST, FDIV{s,d}, FMUL{D}8SUx16, FMUL{D}8ULx16, FMUL{s,d}, FMUL8x16{AL,AU}, FPACK{16,32,FIX}, FsMULd, FSQRT{s,d}

부동소수점/그래픽 명령어를 이슈할 때는 다음의 규칙을 고려한다.

- true dependency : 부동소수점/그래픽 명령어의 결과를 사용하는 명령어는 그 명령어의 결과가 나올 때까지 이슈되지 못한다.
- output dependency : 같은 부동소수점 레지스터에 결과를 쓰는 2 개의 명령어는 같은 사이클에 이슈되지 못한다. 그림 3-15 에 부동소수점/그래픽 명령어의 데이터 의존성에 대한 예가 나와 있다.
- 조건 코드 : 설계상의 간략화를 위해서 FB{P}fcc 명령어는 FCMP{E}{s,d}가 같은 조건 코드를 변경하는 경우뿐 아니라 그렇지 않은 경우에도 앞의 FCMP{E}{s,d}와 같은 사이클에 이슈될 수 없다. 그리고 {F}MOVcc 는 FCMP{E}{s,d}의 결과를 사용할 수 있을 때까지 이슈될 수 없다.
- 그래픽 명령어, FdTOi, FxTOs, FdTOs, FDIVs 와 FSQRTs 의 목적 레지스터가 단정도 이지만 명령어 자체가 배정도 명령어로 취급되기 때문에 그 단정도 레지스터를 포함하는 배정도 레지스터에 대한 데이터 의존성에 영향을 준다.
- FDIV{s,d}와 FSQRT{s,d}는 out-of-order 방식으로 실행 결과를 목적 레지스터에 쓰기 때문에 같은 레지스터를 목적 레지스터로 가지는 명령어는 true dependency 를 갖는 경우와 같이 취급된다.
- 기능 유닛의 충돌: 같은 기능 유닛을 사용하는 2 개의 부동소수점/그래픽 명령어를 같은 사이클에 이슈할 수 없다. 즉 1 개의 A-class 명령어와 1 개의 M-class 명령어만이 단일 사이클에 이슈될 수 있다.

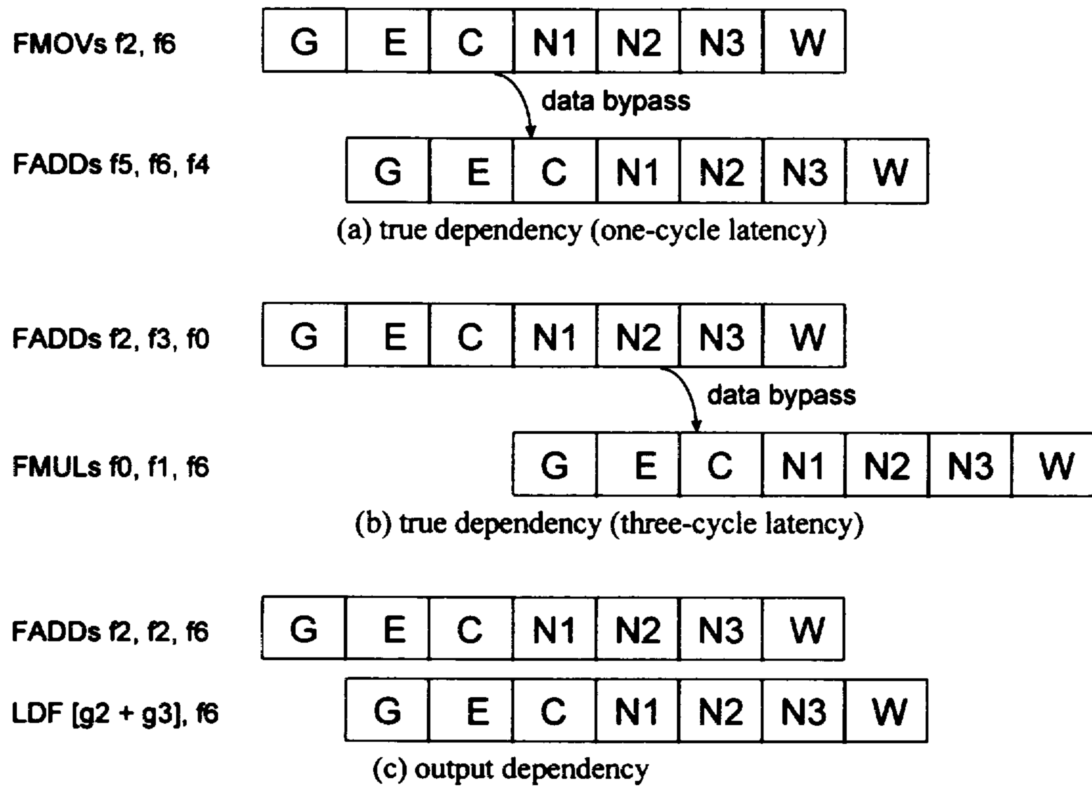


그림 3-15. 부동소수점/그래픽 명령어의 데이터 의존성

(6) 4 배정도(quad precision) 부동소수점 명령어

SPARC-V9 명령어 중에서 4 배정도 부동소수점 명령어는 본 마이크로프로세서가 지원하지 않는다. 따라서 그러한 명령어들은 트랩을 통한 소프트웨어 에뮬레이션(emulation)을 사용하여 처리하여야 한다. 4 배정도 부동소수점 명령어와 그에 따른 트랩의 종류가 표 3-5 에 나와 있다.

그룹화 로직에서는 4 배정도 부동소수점 명령어를 검출하여 그에 따른 오류 신호를 발생시킨다. 그 외에도 명령어 자체에서 검출할 수 있는 오류[7]를 검사한다.

표 3-5. 4 배정도 부동소수점 명령어와 관련 트랩

트랩	명령어
fp_exception_other (unimplemented FPop)	F{s,d}TO, F{i,x}TO, FqTO{s,d}, FCMP{E}q, FMOVq, FMOVqcc, FMOVqr, FABSq, FADDq, FDIVq, FdMULq, FMULq, FNEGq, FSQRTq, FSUBq
illegal_instruction	LDQF{A}, STQF{A}

(7) 그 외의 규칙

그룹화 규칙은 앞에서 설명한 바와 같이 매우 복잡하고 다양하다. 그러나 이 외에도 사소한 규칙이 많이 있는데 이것은 기능 유닛들의 특성이나 데이터 의존성 검출의 간략화를 위한 결과이다. 예를 들어 load 명령어 다음에 SAVE(save caller's window) 명령어를 이슈하게 되면, load 명령어가 로드 버퍼에 들어간 경우 load 명령어에 대한 데이터 의존성을 검사하기가 매우 힘들어진다. 따라서 load 명령어가 이슈된 후부터 load 명령어의 결과 데이터를 받기 전까지는 SAVE 명령어를 이슈할 수 없다.

3.3.3 절 그룹화 로직의 구조

명령어 버퍼는 그림 3-16 에 나와 있듯이 4 개의 기본 블럭으로 이루어졌다. 각 블럭은 명령어 버퍼에서 입력되는 4 개의 명령어 중에서 1 개의 명령어를 3.3.2 절에서 설명한 그룹화 규칙에 따라 이슈한다. 예를 들어 4 개의 명령어 중에서 첫째 명령어의 이슈를 담당하는 블럭(gslot0)은 첫번째 명령어를 입력으로 받아서, 그룹화 규칙에 따라 그 명령어의 이슈 여부를 결정한다. 그리고 이전의 명령어는 다음에 오는 명령어의 이슈에 영향을 미치기 때문에 첫째 명령어 이슈 블럭은 둘째부터 넷째까지의 명령어들을 이슈하기 위해서 필요한 제어 신호를 보내주어야 한다. 예를 들어서 첫째 명령어가 이슈되지 않거나 단일 그룹 명령어인 경우 뒤의 명령어들은 이슈될 수 없으므로 이런 경우를 알려주는 신호(noissue1)를 보내주어야 한다. 그 외에도 데이터 의존성 검사를 위해서 목적 레지스터의 주소를 보내주어야 하고, 기능 유닛의 충돌로 인해서 명령어를

이슈할 수 없는 경우을 검사하기 위해서 어느 기능 유닛을 사용하는 명령인지를 알려주는 신호도 보내주어야 한다. 이러한 신호들 외에도 첫번째 명령어에 대한 정보 중에서 다음 명령어의 이슈에 필요한 것을 알려주는 신호들을 다음 명령어 이슈 블럭에 보내주어야 한다.

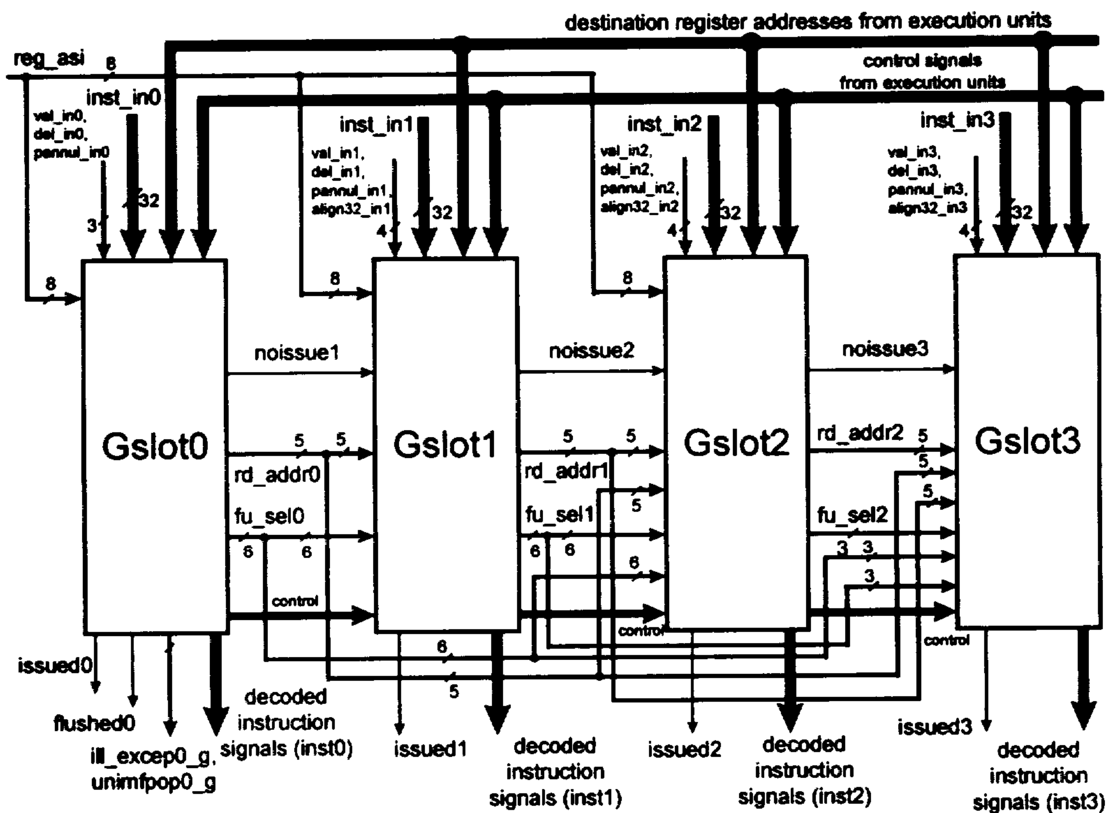


그림 3-16. 그룹화 로직의 블럭다이어그램

첫번째 명령어를 이슈하는 블럭의 내부 구조는 그림 3-17과 같다. 우선 입력된 명령어가 무엇인지를 알기 위해서 명령어의 해석 작업이 블럭의 전단에서 이루어진다. 명령어 해석은 10개의 블럭으로 나뉘어져서 하는데, 표 3-6과 같이 각 블럭은 서로 다른 종류의 명령어를 해석한다. 해석된 신호는 외부로 출력되기도 하지만 그 명령어의 이슈 여부를 판단하는데 사용된다. 명령어를 해석해서 얻은 신호, 명령어와 같이 입력된 신호들, 외부(실행 유닛들)로부터 입력되는 제어 신호들, 명령어의 레지스터 주소들 및 현재 실행 유닛들에서 실행 중인 명령어들의 목적 레지스터 주소들을 이용하여 입력된 명령어의 이슈를 막는 요인이 있는지를 검사한다. 이슈를 막는 원인은 여러 블럭으로 나뉘어서 검사된다. 예를 들어서 현재 실행 유닛에서 어떤 명령어가 실행 중이기 때문에 입력된 명령어를 이슈하지 못하는 경우를 판단하는 블럭, 데이터 의존성을 검사하는 블럭 등 여러 블럭이 있다. 여러 블럭 중에서 정수 명령어의 데이터 의존성을 검사하는 블럭의 내부 구조가 그림 3-18에 한 예로서 나와 있다. 그림 3-18은 첫째 오퍼랜드 레지스터에 의한 데이터 의존성을 검사하는 부분이며 다른 레지스터로 인한 데이터 의존성을 검사하는 것도 비슷한 구조를 가진다. 현재 실행 중인 load 명령어에 대한 데이터 의존성을 검사하기 위해서는 E 단계에 있는 load 명령어의 목적 레지스터에 대한 검사뿐 아니라 로드 버퍼에 있는 명령어들에 대해서도 검사를 하여야 하기 때문에 매우 많은 비교기가 필요하다. 그리고 load 명령어가 정수 레지스터뿐 아니라 부동소수점 레지스터에도 데이터를 쓰는 경우가 있으므로 2가지 경우에 대해서 모두 검사를 해 주어야 하는 어려움이 있다. 부동소수점

명령어에 대한 데이터 의존성 검사의 경우, 부동소수점 명령어의 지연 시간이 다양하기 때문에 각 파이프라인 단계의 목적 레지스터 주소와 입력 명령어의 레지스터 주소를 비교함과 동시에 각 단계에서 어떤 명령어가 실행되고 있는지를 검사하여 종합적으로 데이터 의존성을 검출해야 한다. 특이할 만한 것은 illegal_instruction 오류와 fp_exception_other(unimplemented FPop) 오류를 검사하는 블럭이 있는 것이다. 이슈 과정에서 이러한 검사를 할 수 있는 것은, 이 2종류의 오류는 명령어의 해석에 의해서 알 수 있고, 명령어 자체에 대한 정보를 가장 많이 가지고 있는 그룹화 로직에서 검사하는 것이 적당하기 때문이다. 그 외에 명령어 이슈 여부에는 관계가 없지만 어떤 기능 유닛을 선택할 것인지를 결정하는 블럭이 있다. 이 블럭에 의한 정보는 기능 유닛들에 보내져 적당한 기능 유닛을 선택하는데 사용되며 다음 명령어 이슈 블럭의 명령어 이슈 과정에서 기능 유닛 충돌을 검사하는데 사용된다. 여러 블럭의 검사에 의해서 결정된 이슈 제한 원인들은 그 명령어의 이슈 여부를 최종적으로 판단하는 블럭으로 입력되어 그 명령어의 이슈가 결정된다.

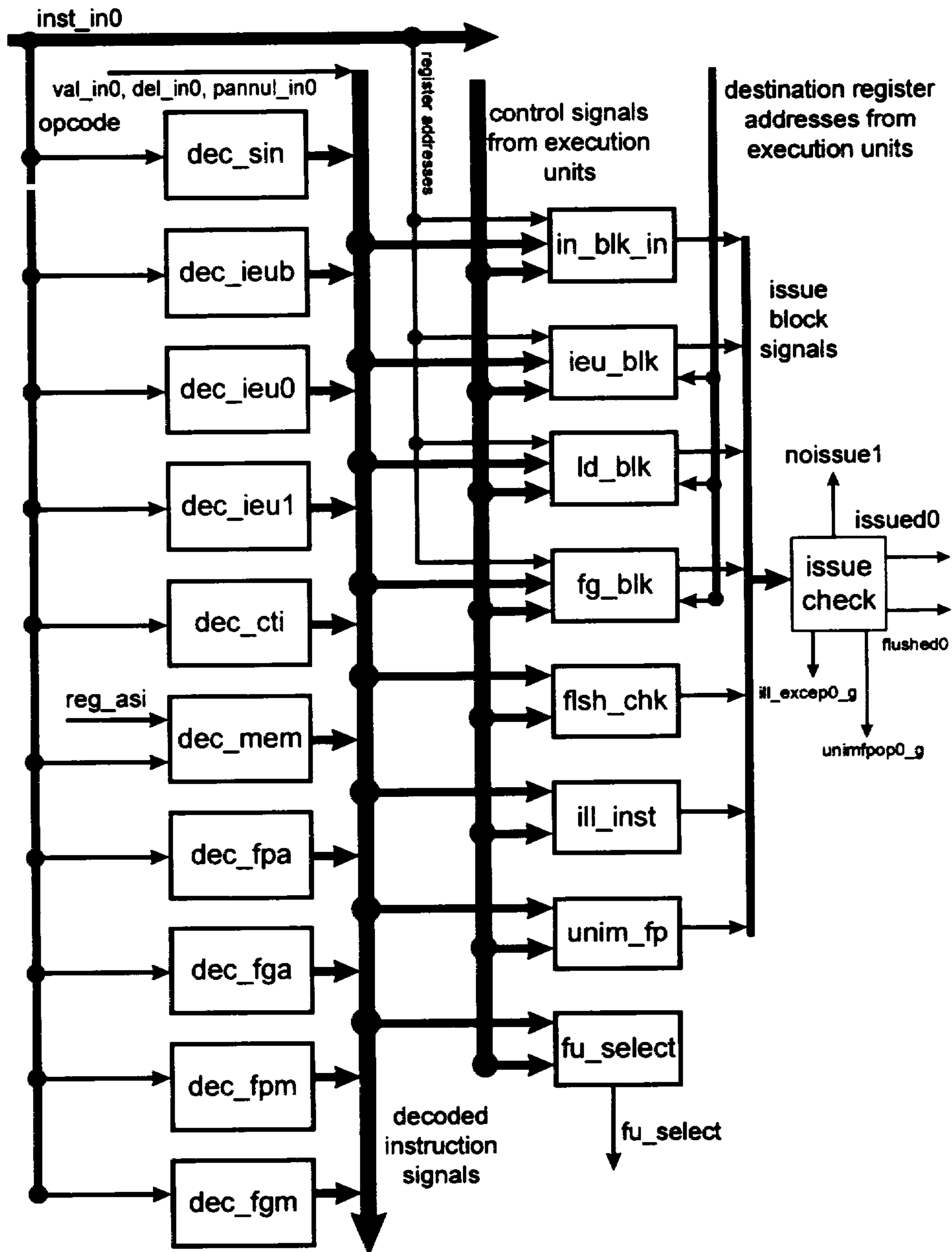


그림 3-17. 첫째 명령어를 이슈하는 블럭의 구조

표 3-6. 명령어 해석 블럭과 해석 명령어 종류

해석 블럭	해석 명령어 종류
dec_sin	단일 그룹 명령어
dec_ieub	불특정 유닛 명령어
dec_ieu0	유닛 0 명령어
dec_ieu1	유닛 1 명령어
dec_cti	분기 명령어
dec_mem	load/store 명령어
dec_fpa	A-class 부동소수점 명령어
dec_fga	A-class 그래픽 명령어
dec_fpm	M-class 부동소수점 명령어
dec_fgm	M-class 그래픽 명령어

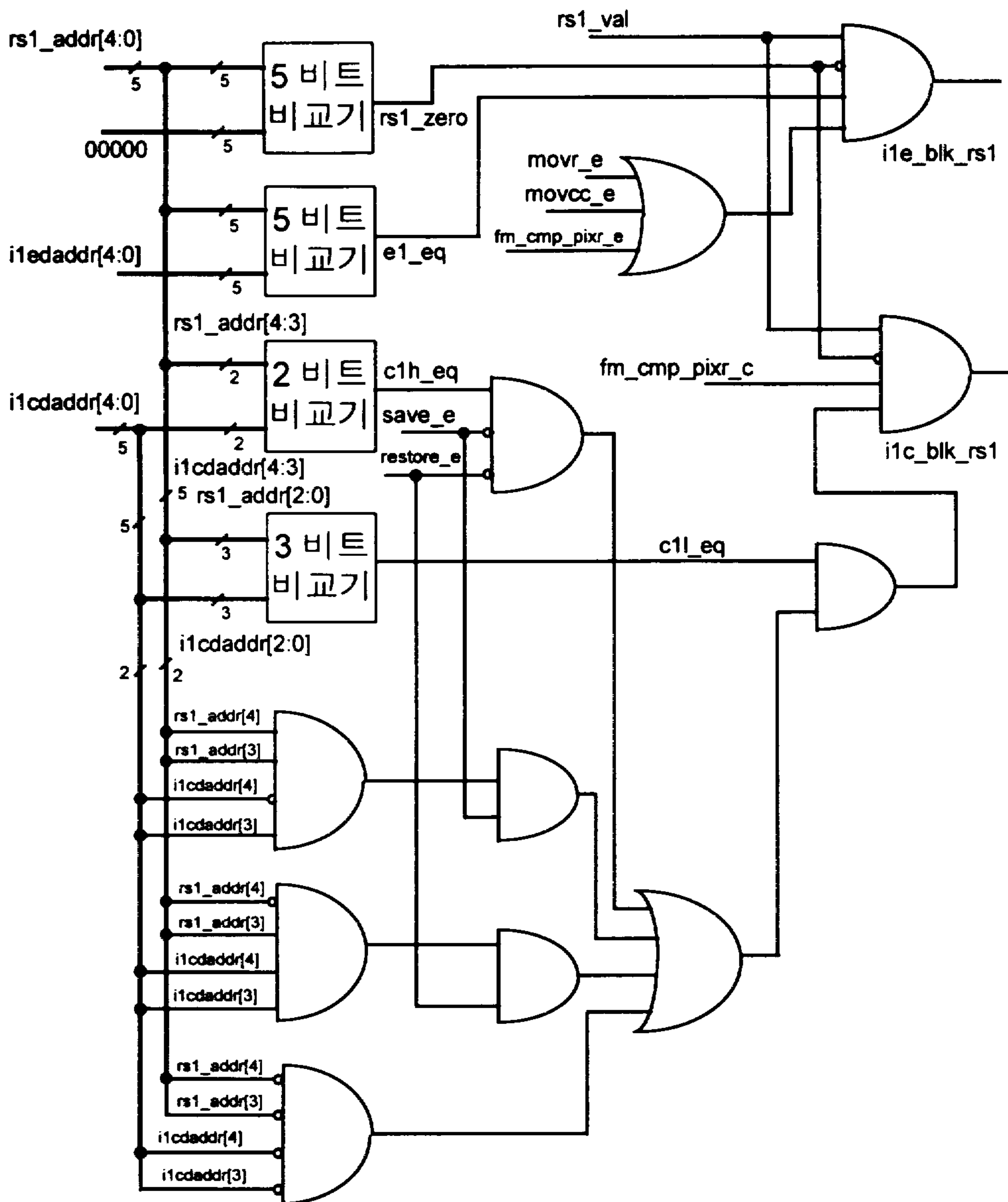


그림 3-18. 정수 명령어 데이터 의존성 검사 로직

둘째 명령어의 이슈를 담당하는 블럭은 첫째 명령어 이슈 블럭과 기본적인 구조는 같으나 사소한 부분에서 달라지는 것이 있다. 우선 둘째 명령어가 단일 그룹 명령어인 경우에 그 명령어는 이슈되지 못하기 때문에 단일 그룹 명령어에 해당하는 명령어 해석 작업이 이루어지지 않는다. 첫째 명령어와 기능 유닛 충돌이 일어나는지를 검사하는 블럭이 추가로 필요하며, 첫째 명령어에 대한 데이터 의존성을 검사하는 부분이 있다. 2 종류의 오류가 검사되기는 하지만 그에 따른 오류 신호를 출력하지는 않고, 단지 그 명령어를 이슈하지 않음으로써, 둘째 명령어가 다음 사이클에서 첫째 명령어가 될 때 오류 신호를 출력하도록 한다. 셋째 명령어를 이슈하는 블럭은 둘째 명령어 블럭과 거의 같다. 단지 차이가 있다면 둘째 명령어에 대한 데이터 의존성과 기능 유닛 충돌을 검사해야 한다는 것이다. 넷째 명령어를 이슈하는 블럭은 앞의 블럭들에 비해 많이 단순해 지는데 그 이유는 정수 명령어와 load/store 명령어가 넷째 명령어로 입력되는 경우 이슈될 수 없기 때문에 이 2 종류의 명령어에 대한 해석 작업을 하지 않아도 되고 이로 인해 블럭의 다른 부분들이 간단해지기 때문이다. 넷째 명령어 이슈 블럭은 이러한 것을 제외하고는 셋째 명령어 블럭과 거의 같다.

제 4 장 모의실험 및 검증

본 논문의 이슈 유닛을 검증하기 위해서 이슈 유닛을 HDL을 이용하여 기술하고 모의실험을 수행하였다. 명령어 정렬기, 명령어 버퍼 및 그룹화 로직을 behavioral 및 structural 수준에서 기술하고, 그 모델들의 모의실험을 수행하여 각 블럭의 검증을 한 후 모든 블럭을 통합해서 전체 이슈 유닛에 대한 검증을 하였다.

4.1 절 하위 블럭의 검증

명령어 정렬기는 게이트와 멀티플렉서 및 플립플롭(flip-flop)을 이용하여 structural 수준에서 기술되었으며, 시뮬레이션 모듈에 의해서 발생된 입력 벡터를 명령어 정렬기의 HDL 모델에 입력하고, 그에 따른 신호들의 파형을 관찰함으로써 검증을 수행하였다. 입력 벡터는 모든 경우의 오류를 검출하기 위해서 다양하게 발생시켰으며, 특히 유효 비트의 모든 조합에 대해서 검증을 함으로써 명령어 정렬기가 올바르게 동작하는지를 확인하였다. 부록 1.1에 명령어 정렬기의 검증 파형이 나와 있다.

명령어 버퍼는 하위 블럭의 HDL 모델을 만들어 동작 검증을 한 후 블럭들을 통합하여 명령어 버퍼 전체의 동작을 검증하였다. 명령어 정렬기의 검증과 같이 시뮬레이션 모듈을 별도로 만들었으며, 이 모듈에 의해 입력 벡터를 명령어 버퍼의 HDL 모델에 입력시킨 후에 파형 관찰을 통해 검증을 수행하였다. 그 결과 파형의 일부가 예로서 부록 1.2에 나와

있다. 완벽한 검증이 되도록 하기 위해서 입력 유효 비트와 issued 비트의 다양한 조합에 대해서 모의실험을 수행하였으며, 특히 명령어 저장 어레이에 명령어를 저장할 충분한 공간이 없는 경우의 동작을 관찰하기 위해 이 경우에 대한 다양한 입력 벡터를 사용하였다. 명령어 저장 어레이의 내부 상태를 매 사이클마다 관찰함으로써 내부 동작의 적당성을 검증하였으며, 외부 동작의 검증은 외부 출력 신호를 관찰함으로써 알 수 있었다.

그룹화 로직은 실제적인 이슈 기능을 수행하는 블럭이기 때문에 매우 복잡한 로직으로 되어있으며 많은 하위 블럭들을 포함한다. 우선 하위 모듈들을 기술하고 검증한 후 통합하여 그룹화 로직의 검증을 수행하였다. 다중 사이클 명령어에 의해 명령어 이슈가 정지되는 경우의 모의 실험 결과 파형이 부록 1.3 에 나와 있다. 그룹화 로직은 입력되는 명령어 뿐 아니라 실행 유닛에서 실행되고 있는 명령어에 따라서 다르게 동작하므로 입력 명령어와 실행 유닛으로부터의 신호를 다양하게 하면서 검증을 수행하였다. 실행 유닛들은 아직 설계되지 않았기 때문에 실행 유닛으로부터 입력되는 신호들은 검증의 완벽성을 고려해서 임의로 발생시켰으며 출력되는 모의실험 파형을 관찰함으로써 동작의 적당성을 검증하였다.

그림 4-1 에 하위 블럭의 모의실험 과정이 나와 있다.

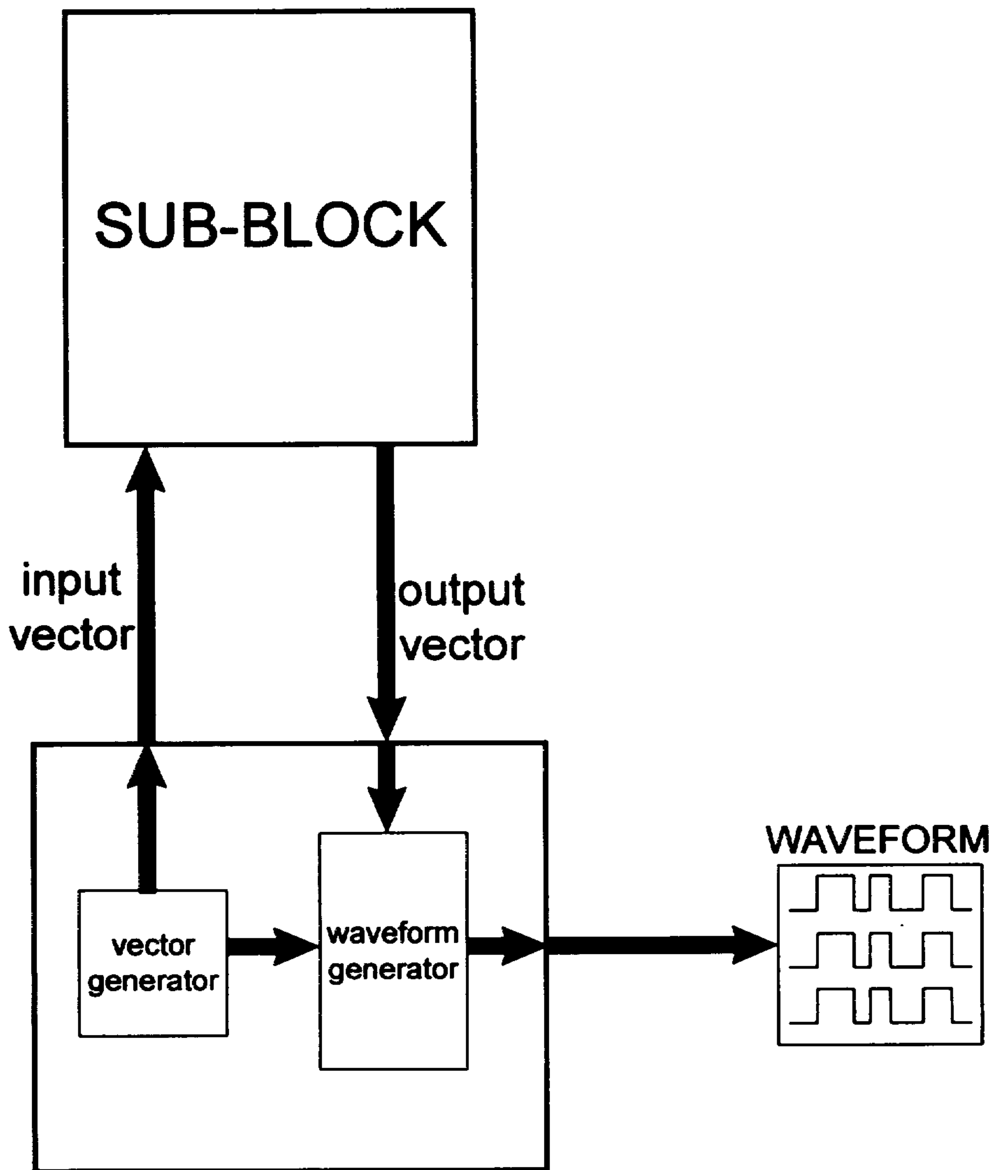


그림 4-1. 하위 블럭의 검증

4.2 절 이슈 유닛의 검증

블럭별로 검증을 수행한 후 통합 모듈을 만들어 이슈 유닛 전체에 대한 검증을 수행하였다. 부록 3 에 이슈 유닛의 HDL top module 이 예로서 나와 있다. 입력 벡터를 만들기 위해서 우선 C 프로그램을 gcc[24][25]로 컴파일(compile)하여 만든 실행 파일을 hex 파일로 바꾼 후 프로그램의 특성을 나타내는 일부분을 이슈 유닛의 입력에 맞는 형식으로 변형시켰다. 실행 유닛으로 입력되는 신호들은 입력 명령어 열을 미리 분석하여 시뮬레이션 모듈에서 발생시키게 하였다. 이렇게 만들어진 입력 벡터를 이용하여 모의실험을 수행하여 검증 및 성능 평가를 하였다. 사용된 프로그램은 hanoi tower, sieve, quick sort, matrix 및 bubble sort 이며, 각 프로그램에서 hex 파일을 하나씩 만들어 모의실험을 수행하였다. 사용된 hanoi tower 프로그램의 hex 코드 및 어셈블리 코드가 예로서 부록 2 에 나와 있으며, 이 명령어 열을 사용한 모의실험의 파형이 부록 1.4 에 나와 있다. 모의실험 과정에서는 명령어 캐쉬와 데이터 캐쉬의 히트율과 분기 예측의 정확성을 100%로 가정하였으며, 그림 4-2 와 그림 4-3 에 이슈 유닛의 모의실험 과정이 나와 있다. 표 4-1 에 성능 평가 결과가 나와 있으며, 표 4-1 에서 알 수 있듯이 약 1.8 IPC 의 성능을 보여주었다.

표 4-1. C 프로그램 수행 결과

	src1	src2	src3	src3	src4	전체
총 이슈 명령어 수	54	51	53	53	55	266
총 수행 사이클 수	30	28	28	31	31	148
0-issue 사이클 수	2	1	2	3	2	10
1-issue 사이클 수	12	11	8	11	10	52
2-issue 사이클 수	6	8	10	10	12	46
3-issue 사이클 수	10	8	7	6	7	38
4-issue 사이클 수	0	0	1	1	0	2
IPC	1.8	1.821	1.929	1.710	1.774	1.797
프로그램	hanoi tower	sieve	quick sort	matrix	bubble sort	-

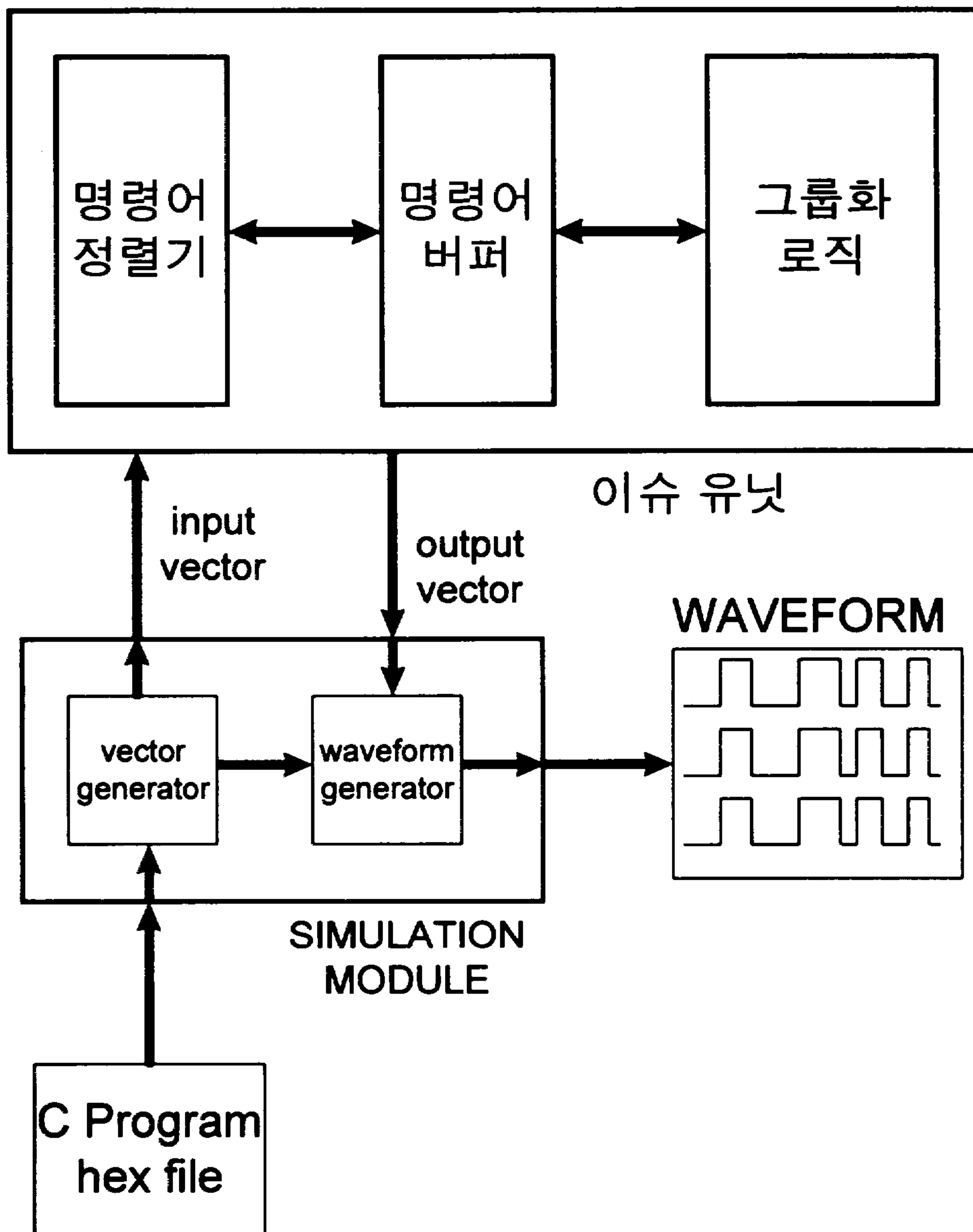


그림 4-2. C 프로그램을 이용한 이슈 유닛의 검증

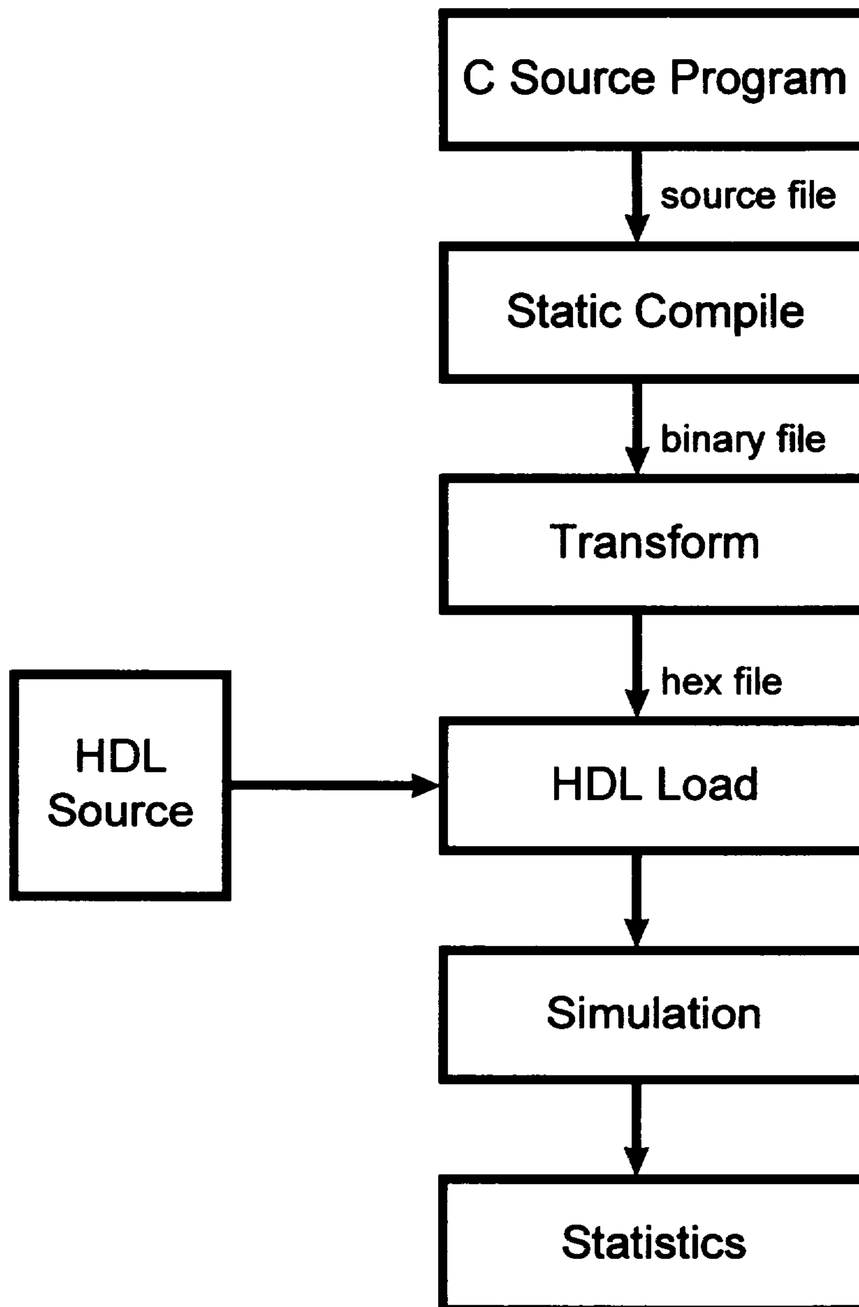


그림 4-3. 성능 평가 과정

SPARC 명령어 세트를 사용하는 3-way 수퍼스칼라 마이크로프로세서인 SuperSPARC의 1.4 IPC[4]와 비교해서 본 이슈 구조는 개선된 성능을 보여준다. 그리고 1개, 2개 또는 3개의 명령어를 이슈하는 사이클이 가장 많고, 0개 또는 4개의 명령어를 이슈하는 사이클이 매우 드물다. 0-issue 사이클은 대부분 2 사이클의 지연 시간을 갖는 load 명령어에 대한 데이터 의존성으로 인한 인터락 때문에 발생하였으며, 명령어를 1개나 2개만 이슈하는 사이클은 같은 사이클에 G 단계에 존재하는 이전 명령어에 대한 데이터 의존성이 주요 원인이었다. 4-issue 사이클이 극히 드문 것은 프로그램의 명령어들이 넷째 명령어로는 이슈할 수 없는 정수 명령어와 load/store 명령어 위주로 되어있고, SuperSPARC 마이크로프로세서용 컴파일러를 사용하였기 때문이다. 다시 말해서 부동소수점 명령어와 분기 명령어만이 넷째 명령어로서 이슈될 수 있기 때문에 부동소수점 명령어가 없는 프로그램을 수행할 때는 성능이 낮아진다. 그러나 load와 load의 결과를 사용하는 명령어 사이에 충분한 간격을 둘 수 있거나, 부동소수점 명령어를 포함한 프로그램을 본 마이크로프로세서에 최적화된 컴파일러로 컴파일한 프로그램을 사용한다면 더욱 좋은 성능을 가질 것이다.

제 5 장 결 론

본 연구에서는 SPARC version 9 명령어 세트를 채택한 4-way 수퍼스칼라 마이크로프로세서의 이슈 유닛을 HDL을 이용하여 모델링하였다.

본 이슈 유닛은 기본적인 명령어 세트에 그래픽 처리를 위한 명령어를 추가함으로써 그래픽 데이터를 처리하는 프로그램의 수행 속도를 향상시키고자 했다. 비용과 성능을 고려하여 실행 유닛과 레지스터파일과 같은 자원의 수 및 구조를 결정하였으며, 이를 바탕으로 최적화된 구조로 이슈 유닛을 설계하였다. 이슈 유닛은 충분한 명령어를 제공받기 위해서 페치된 명령어를 명령어 버퍼에 일시적으로 저장한 후 이슈함으로써 명령어의 페치와 이슈를 분리시켰으며, 또한 명령어를 정렬한 후 이슈함으로써 이슈 유닛을 충분히 활용하게 하였다. 명령어들은 순차적 방식으로 단일 사이클에 최대 4개까지 이슈될 수 있으며, 명령어 이슈 규칙은 실행 유닛의 수와 레지스터파일의 포트 수를 고려해서 최적의 것을 채택하였다.

이슈 유닛은 명령어 정렬기, 명령어 버퍼 및 그룹화 로직으로 이루어졌으며, 프리페치 유닛에서 입력되는 명령어가 블럭들을 통해 순서대로 흘러가다가 실행 유닛으로 이슈되도록 되어있다. 명령어 정렬기는 처음으로 유효한 명령어가 첫째 명령어로 출력되도록 명령어의 순서를 조정하는 역할을 하며, 명령어 버퍼는 명령어를 정렬하여 저장함으로써 그룹화 로직에 충분한 명령어들을 효율적으로 제공하는 역할을 한다. 그리고 그룹화 로직은 명령어 버퍼로부터 입력되는

명령어들을 최적화된 규칙에 따라 순서대로 실행 유닛에 이슈하는 기능을 한다.

이슈 유닛은 behavioral 수준 및 structural 수준에서 HDL로 모델링하여 모의실험을 수행하여 검증 및 성능 평가를 하였다. 모의실험의 입력을 만들기 위해서 C 프로그램을 컴파일하여 만들어진 실행 파일을 hex 파일로 바꾼 후 입력 벡터의 형식으로 변경하였다. 실행 유닛으로부터의 입력은 명령어 열의 분석에 의해 모의실험을 수행하기 전에 만들어서 이슈 유닛에 제공하였다. 명령어 캐쉬와 데이터 캐쉬의 히트율과 분기 예측의 정확성이 100%인 경우 5개의 프로그램을 사용하여 성능 평가를 한 결과 IPC는 1.8로 측정되었으나, 부동소수점 명령어를 포함하는 프로그램을 최적화된 컴파일러를 이용하여 컴파일해서 모의실험을 수행할 경우 더 좋은 성능을 가질 것으로 추정된다.

본 이슈 유닛의 HDL 모델은 로직 수준의 설계를 고려하여 자동합성(synthesis)[26]이 가능하도록 기술되었으므로, 이후에는 다른 유닛들과 통합하여 자동합성한 후 최적화된 layout을 하는 작업이 필요하다.

참고 문헌

[1] Kai Hwang, *Advanced Computer Architecture: Parallelism, Scalability, Programmability*, McGraw-Hill, pp. 308-322, 1993.

[2] Mike Johnson, *Superscalar Microprocessor Design*, Prentice-Hall, pp. 9-175, 1991.

[3] Michael J. Flynn, *Computer Architecture: Pipelined And Parallel Processor Design*, Jones and Bartlett Publishers, pp. 456-498, 1995.

[4] SPARC Technology Business, *SuperSPARC Family STP1020 & STP1090 Series User's Manual*, Sun Microsystems, Inc., revision 1.C, 1994.

[5] Robert A. Iannucci, Guang R. Gao, Robert H. Halstead, Jr., Burton Smith, *Multithreaded Computer Architecture: A Summary of The State of The Art*, Kluwer Academic Publishers, pp. 1-60, 1994.

[6] Ben Catanzaro, *Multiprocessor System Architectures*, Sun Microsystems, Inc., pp. 4-46, 1994.

[7] David L. Weaver and Tom Germond, *The SPARC Architecture Manual*, Prentice-Hall, Inc., version 9, 1994.

[8] Boney, Joel, "SPARC Version 9 Points the Way to the Next Generation RISC", *Sunworld*, pp. 100-105, Oct. 1992.

[9] Ted Williams, Niteen Patkar, and Gene Shen, "SPARC64: A 64-b 64-Active-Instruction Out-of-Order-Execution MCM Processor", *IEEE Journal of Solid-State Circuits*, vol. 30, pp. 1215-1225, Nov. 1995.

[10] SPARC Technology Business, *UltraSPARC™-I User's Manual*, Sun Microsystems, Inc., revision 1.0, 1995.

[11] Lavi A. Lev et al., "A 64-b Microprocessor with Multimedia Support", *IEEE Journal of Solid-State Circuits*, vol. 30, pp. 1227-1238, Nov. 1995.

[12] D. Greenley et al., "UltraSPARC(TM): The Next Generation Superscalar 64-bit SPARC", *40th annual Compcon*, Compcon, pp. 442-451, 1995.

[13] SPARC Technology Business, *UltraSPARC-I Data Sheet*, Sun Microsystems, Inc., 1995.

[14] Marc Tremblay and J. Michael O'Connor, "UltraSPARC I: A Four-Issue Processor Supporting Multimedia", *IEEE Micro*, pp. 42-49, Apr. 1996.

[15] Shlomo Weiss and James E. Smith, "Instruction Issue Logic in Pipelined Supercomputers", *IEEE Transactions on Computers*, vol. c-33, pp. 1013-1022, Nov. 1984.

[16] Gurindar S. Sohi, "Instruction Issue Logic for High-Performance, Interruptible, Multiple Functional Unit, Pipelined Computers", *IEEE Transactions on Computers*, vol. 39, pp. 349-359, Mar. 1990.

[17] John L. Hennessy and David A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers, Inc., pp. 290-337, 1990.

[18] L. Kohn et al., "The Visual Instruction Set (VIS) in UltraSPARC™, *40th annual Compcon*, pp. 462-469, 1995.

[19] Chang-Guo Zhou et al., "MPEG Video Decoding with the UltraSPARC Visual Instruction Set", *40th annual Compcon*, pp. 470-475, 1995.

[20] Manolis G. H. Katevenis, *Reduced Instruction Set Computer Architectures for VLSI*, The MIT Press, pp. 42-85, 1985.

[21] James E. Smith and Andrew R. Pleszkun, "Implementing Precise Interrupts in Pipelined Processors", *IEEE Transactions on Computers*, vol. 37, pp. 562-573, May 1988.

[22] Garold S. Tjaden and Michael J. Flynn, "Detection And Parallel Execution of Independent Instructions", *IEEE Transactions on Computers*, vol. c-19, pp. 889-895, Oct. 1970.

[23] Ramon D. Acosta, Jacob Kjelstrup, and H. C. Torng, "An Instruction Issuing Approach to Enhancing Performance in Multiple Functional Unit Processors", *IEEE Transactions on Computers*, vol. c-35, pp. 815-828, Sep. 1986.

[24] Richard P. Paul, *SPARC Architecture, Assembly Language Programming, and C*, Prentice-Hall, 1991.

[25] Sun Microsystems, Inc., *C-compiler(Sun Release 4.1) on-line manual*, 1988.

[26] COMPASS Design Automation, *ASIC Synthesizer For Verilog Design*, 1991.

부 록

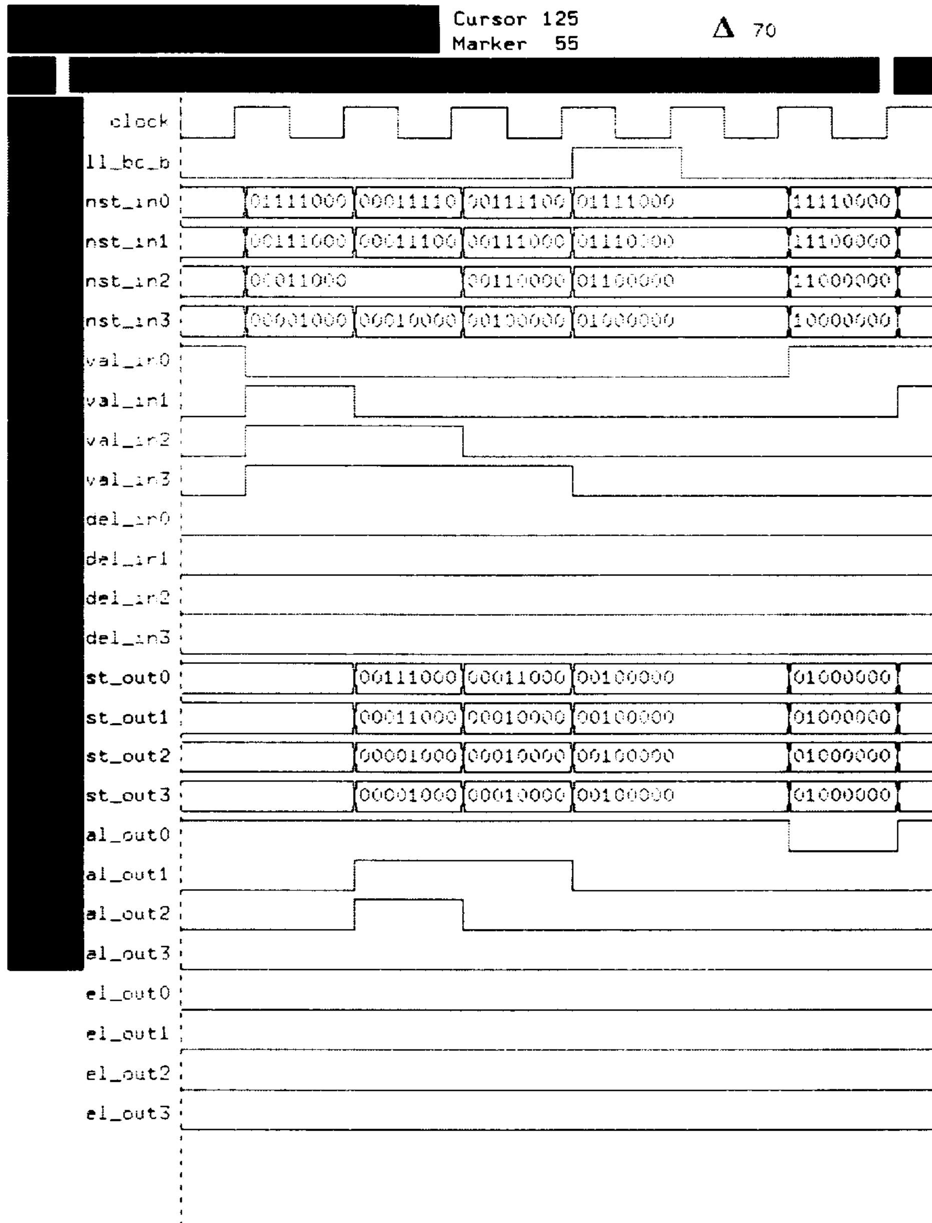
부록 1. HDL 모의실험

부록 2. src1 의 hex code 와 assembly code

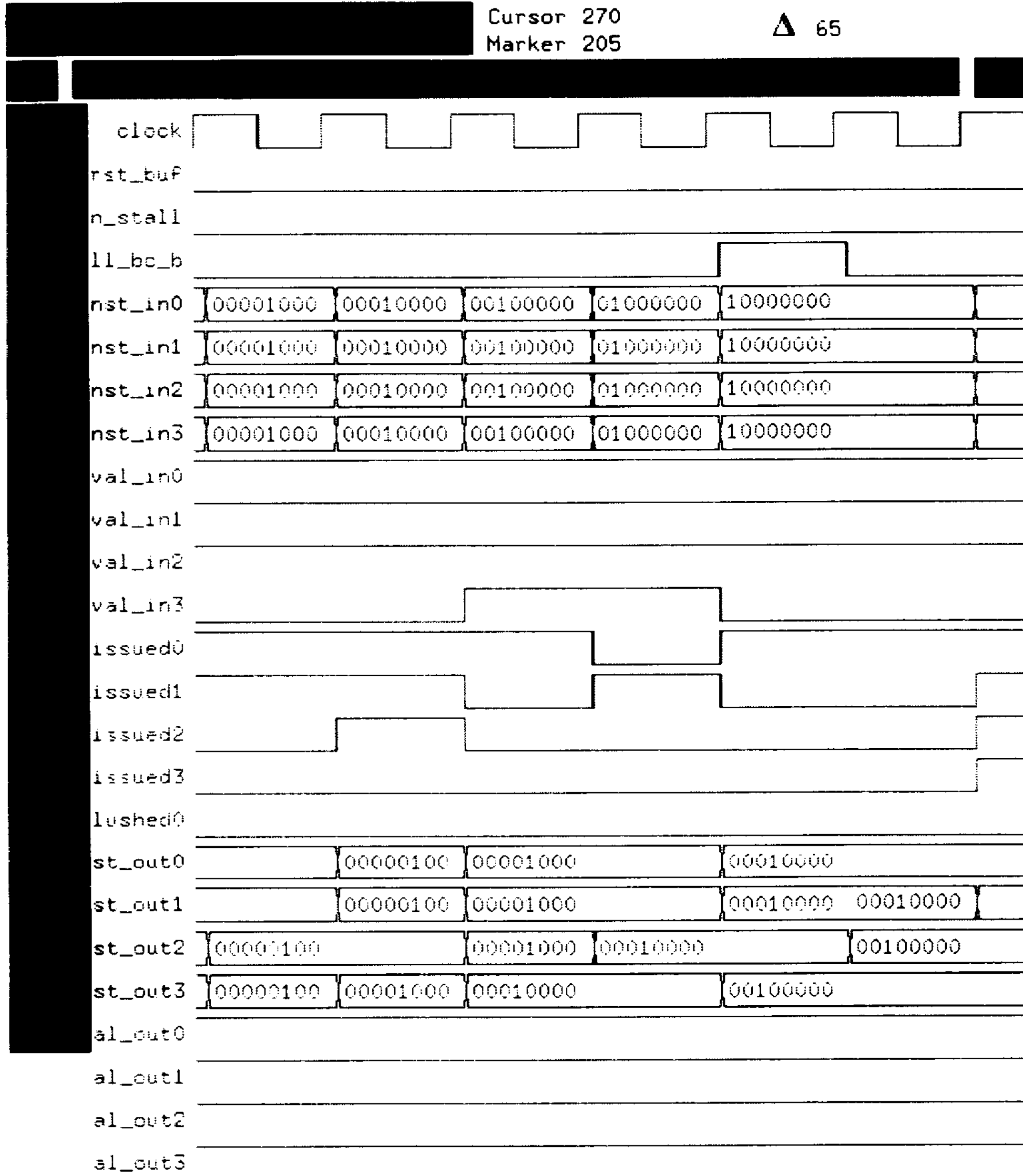
부록 3. 이슈 유닛의 HDL top module

부록 1. HDL 모의실험

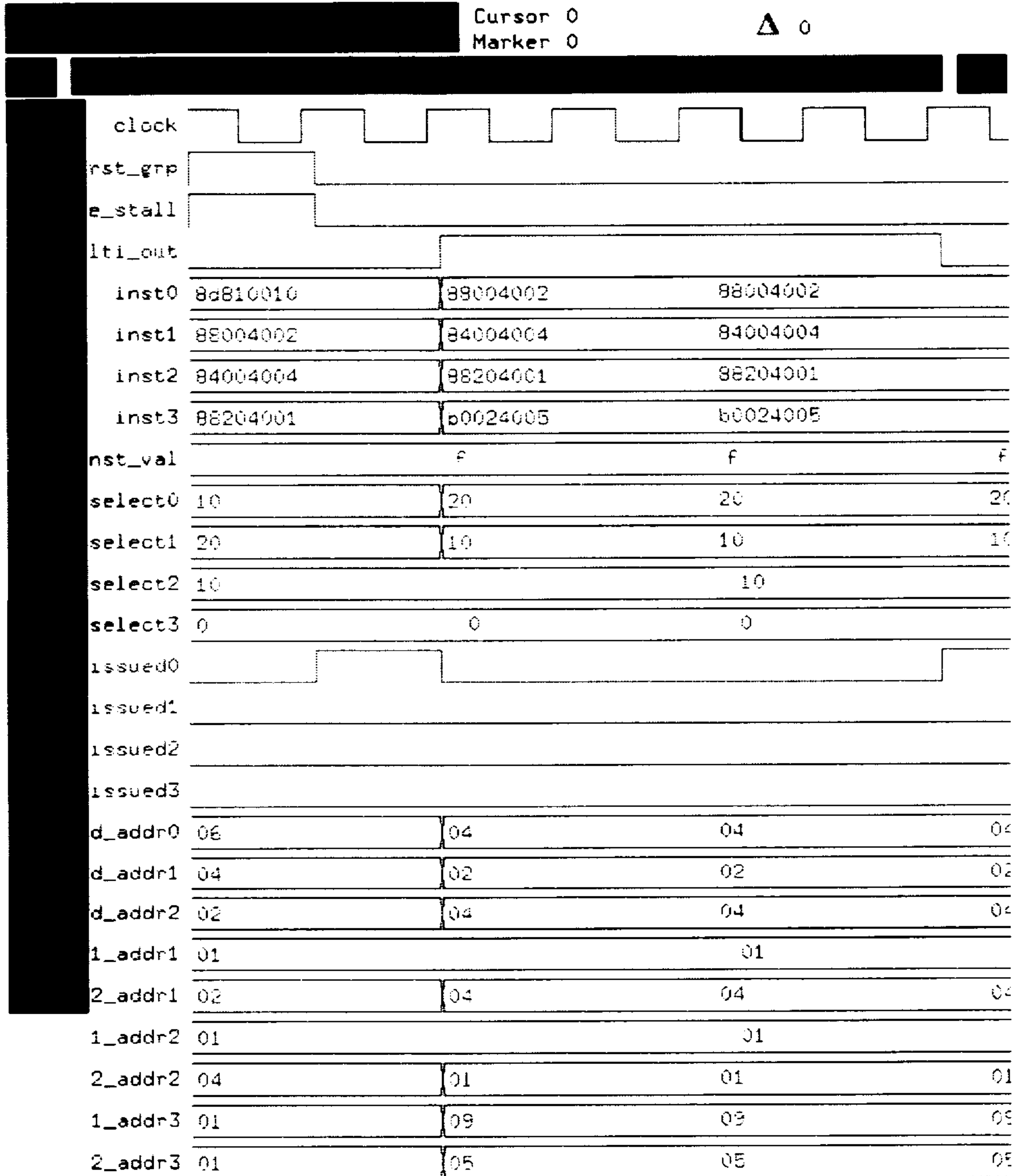
부록 1.1 명령어 정렬기의 검증



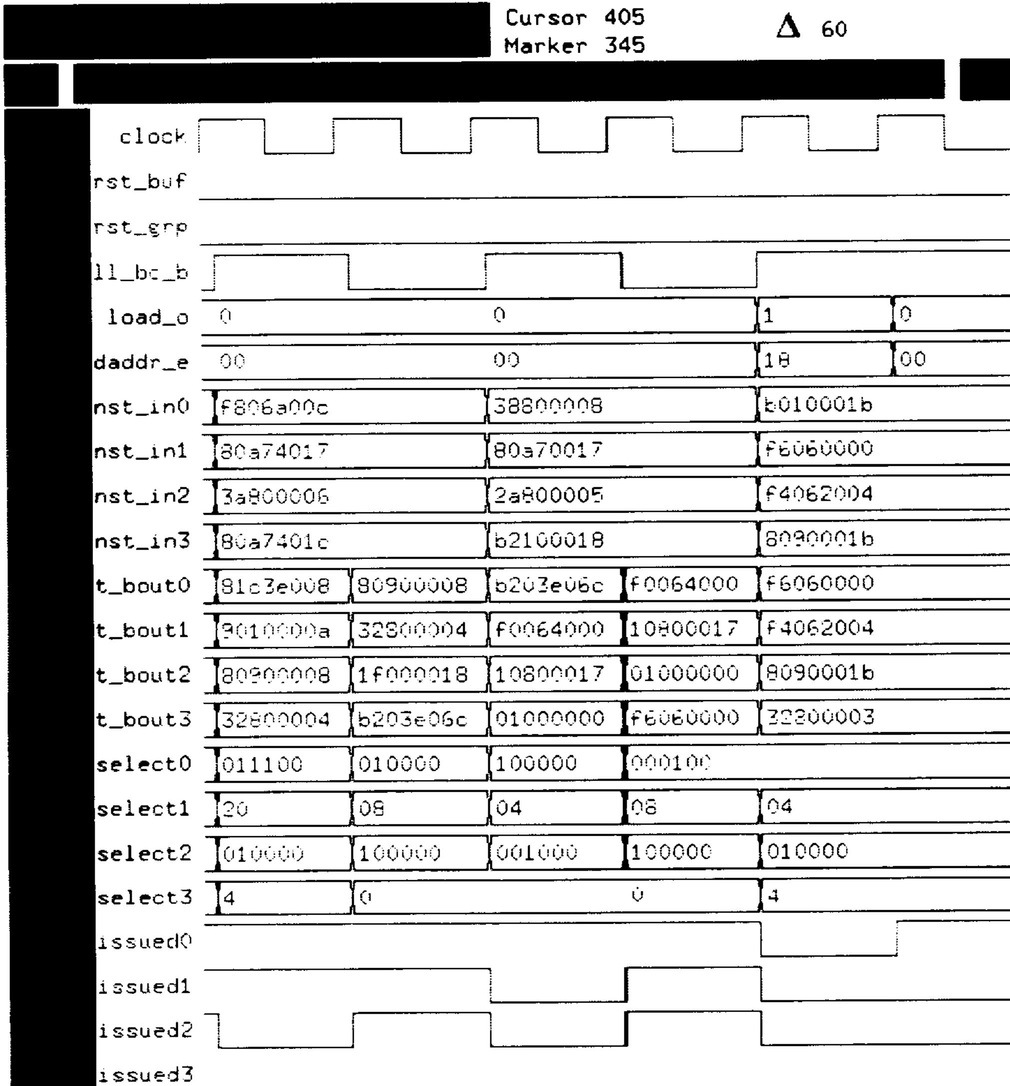
부록 1.2 명령어 버퍼의 검증



부록 1.3 그룹화 로직의 검증



부록 1.4 src1 을 이용한 이슈 유닛의 모의실험 파형



부록 2. src1 의 hex code 와 assembly code

```
/* 0x2d24 */ 400001a8   //  call    0x33c4
/* 0x2d28 */ 90100017   //  mov     %l7, %o0
/* 0x33c4 */ 9de3bfa0   //  save    %sp, 0xffffffa0, %sp
/* 0x33c8 */ 11000018   //  sethi   %hi(0x6000), %o0
/* 0x33cc */ d0022028   //  ld      [%o0+0x28], %o0   ! 0x6028
/* 0x33d0 */ 80900008   //  tst     %o0
/* 0x33d4 */ 3280000e   //  bnz,a   0x340c
/* 0x33d8 */ 21000018   //  sethi   %hi(0x6000), %l0
/* 0x340c */ d0042028   //  ld      [%l0+0x28], %o0   ! 0x6028
/* 0x3410 */ d2042028   //  ld      [%l0+0x28], %o1
/* 0x3414 */ 90222001   //  dec     %o0
/* 0x3418 */ 40000198   //  call    0x3a78
/* 0x341c */ 90060008   //  add     %i0, %o0, %o0
/* 0x3a78 */ 1080000b   //  b       0x3aa4
/* 0x3a7c */ 82102000   //  clr     %g1
/* 0x3aa4 */ 9a924000   //  orcc    %o1, %g0, %o5
/* 0x3aa8 */ 12800004   //  bnz     0x3ab8
/* 0x3aac */ 96100008   //  mov     %o0, %o3
/* 0x3ab8 */ 80a2c00d   //  cmp     %o3, %o5
/* 0x3abc */ 0a800094   //  blu     0x3d0c
/* 0x3ac0 */ 94102000   //  clr     %o2
/* 0x3d0c */ 80900001   //  tst     %g1
/* 0x3d10 */ 26800002   //  bl,a    0x3d18
/* 0x3d14 */ 9420000a   //  neg     %o2
/* 0x3d18 */ 81c3e008   //  retl
/* 0x3d1c */ 9010000a   //  mov     %o2, %o0
```

```
/* 0x3420 */  80900008   //  tst     %o0
/* 0x3424 */  32800004   //  bnz,a   0x2d40
/* 0x3428 */  1f000018   //  sethi   %hi(0x6000), %o7
/* 0x24d0 */  b203e06c   //  add     %o7, 0x6c, %i1
/* 0x2d44 */  f0064000   //  ld      [%i1], %i0
/* 0x2d48 */  10800017   //  b       0x2da4
/* 0x2d4c */  01000000   //  nop
/* 0x2da4 */  f6060000   //  ld      [%i0], %i3
/* 0x2da8 */  f4062004   //  ld      [%i0+0x4], %i2
/* 0x2dac */  8090001b   //  tst     %i3
/* 0x2db0 */  32800003   //  bnz,a   0x2dbc
/* 0x2db4 */  fa06e00c   //  ld      [%i3+0xc], %i5
/* 0x2dbc */  8090001a   //  tst     %i2
/* 0x2dc0 */  32bfffe4   //  bnz,a   0x2d50
/* 0x2dc4 */  f806a00c   //  ld      [%i2+0xc], %i4
/* 0x2d50 */  80a74017   //  cmp     %i5, %l7
/* 0x2d54 */  3a800006   //  bcc,a   0x2d6c
/* 0x2d58 */  80a7401c   //  cmp     %i5, %i4
/* 0x2d6c */  38800008   //  bgu,a   0x2d8c
/* 0x2d70 */  80a70017   //  cmp     %i4, %l7
/* 0x2d8c */  2a800005   //  blu,a   0x2da0
/* 0x2d90 */  b2100018   //  mov     %i0, %i1
/* 0x2da0 */  b010001b   //  mov     %i3, %i0
/* 0x2da4 */  f6060000   //  ld      [%i0], %i3
/* 0x2da8 */  f4062004   //  ld      [%i0+0x4], %i2
/* 0x2dac */  8090001b   //  tst     %i3
/* 0x2db0 */  32800003   //  bnz,a   0x2dbc
/* 0x2db4 */  fa06e00c   //  ld      [%i3+0xc], %i5
```

부록 3. 이슈 유닛의 HDL top module

```
module issue (
	clock, rst_buf, rst_grp, bin_stall, issue_stall, inst_in0, inst_in1, inst_in2, inst_in3, val_in0,
	val_in1, val_in2, val_in3, del_in0, del_in1, del_in2, del_in3, pannul_in0, pannul_in1,
	pannul_in2, pannul_in3, align32_in0, align32_in1, align32_in2, align32_in3, reg_asi, e_stall,
	flush_ads, pre_anuld_e, ieu0_out, ieu1_out, cti_out, mem_out, fa_out, fm_out, multi_out,
	fm_cmp_pixr_e, fm_cmp_pixr_c, fm_cmp_pixr_n1, ieu0_e, ieu1_e, call_e, call_c, jmpl_e,
	jmpl_c, return_e, return_c, bicc_e, bicc_c, bicc_n1, bpcc_e, bpcc_c, bpcc_n1, bpr_e, bpr_c,
	bpr_n1, fbfcc_e, fbfcc_c, fbfcc_n1, fbpfcc_e, fbpfcc_c, fbpfcc_n1, annulb_e, annulb_c,
	annulb_n1, ld_blk_two, ldfsr_b5adr, load_b3an3, load_ndr, load_b3adr, load_9out, store_8out,
	store_out, stfsr_e, stfsr_c, storef_e, st_blk_out, fm_divsqrt_3bf, movcc_e, movr_e, loadi_e,
	loadf_e, fa_addsub_e, fa_addsub_c, fa_addsub_n1, fa_convert_e, fa_convert_c, fa_convert_n1,
	fa_cmp_e, fa_cmp_c, fa_cmp_n1, fa_cmpcc_e, fm_mul_e, fm_mul_c, fm_mul_n1,
	fm_divsqrts_b12, fm_divsqrts_b13, fm_divsqrtd_b22, fm_divsqrtd_b23, fa_man_e, fa_man_c,
	fa_mov_e, fa_mov_c, fa_mov_n1, fa_pasrame_e, fa_pasrame_c, fa_pasrame_n1, fm_pack_e,
	fm_pack_c, fm_pack_n1, fm_pmuldist_e, fm_pmuldist_c, fm_pmuldist_n1, fa_ds_e, fa_ds_c,
	fa_ds_n1, fm_ds_e, fm_ds_c, fm_ds_n1, fa_cds_e, fa_cds_c, fa_cds_n1, fa_daddr_e, fa_daddr_c,
	fa_daddr_n1, fm_daddr_e, fm_daddr_c, fm_daddr_n1, fm_daddr_ds, i1_daddr_e, i1_daddr_c,
	save_e, restore_e, l0_ndr, l1_ndr, l2_ndr, l3_ndr, l4_ndr, l5_ndr, l6_ndr, l7_ndr, l8_ndr, l0_int,
	l1_int, l2_int, l3_int, l4_int, l5_int, l6_int, l7_int, l8_int, l0_fp, l1_fp, l2_fp, l3_fp, l4_fp, l5_fp,
	l6_fp, l7_fp, l8_fp, ld_daddr_e, l0_daddr, l1_daddr, l2_daddr, l3_daddr, l4_daddr, l5_daddr,
	l6_daddr, l7_daddr, l8_daddr, stall_bc_b, issued0, issued1, issued2, issued3, tnd_dctic0,
	pre_anu_d0, pre_anu_d1, pre_anu_d2, pre_anu_d3, ill_excep0_g, unimfpop0_g, slot0_ieu0,
	slot1_ieu0, slot2_ieu0, slot0_ieu1, slot1_ieu1, slot2_ieu1, slot0_mem, slot1_mem, slot2_mem,
	slot0_cti, slot1_cti, slot2_cti, slot3_cti, slot0_fa, slot1_fa, slot2_fa, slot3_fa, slot0_fm, slot1_fm,
	slot2_fm, slot3_fm, seticc0_g, seticc1_g, seticc2_g, a_field0, a_field1, a_field2, a_field3,
	i_field0, i_field1, i_field2, i_field3, x_field0, x_field1, x_field2, p_field0, p_field1, p_field2,
	p_field3, fprnd_sbit0, fprnd_sbit1, fprnd_sbit2, fprnd_sbit3, alt_field0, alt_field1, alt_field2,
	mmask0, mmask1, mmask2, cmask0, cmask1, cmask2, b0_cc, b1_cc, b2_cc, b3_cc, t0_cc, m0_cc,
	opf0_cc, c0_cc, c1_cc, c2_cc, c3_cc, b0_cond, b1_cond, b2_cond, b3_cond, m0_cond, b0_rcond,
	b1_rcond, b2_rcond, m0_rcond, rs1_addr0, rs2_addr0, rd_addr0, rs1_addr1, rs2_addr1, rd_addr1,
```

rs1_addr2, rs2_addr2, rd_addr2, rs1_addr3, rs2_addr3, rd_addr3, asi_a0, asi_a1, asi_a2, inst_bout0, inst_bout1, inst_bout2, inst_bout3);

align a_unit(

clock, stall_bc_b, inst_in0, inst_in1, inst_in2, inst_in3, val_in0, val_in1, val_in2, val_in3,del_in0, del_in1, del_in2, del_in3, pannul_in0, pannul_in1, pannul_in2, pannul_in3, align32_in0, align32_in1, align32_in2, align32_in3, inst_dout0, inst_dout1, inst_dout2, inst_dout3, val_dout0, val_dout1, val_dout2, val_dout3, del_dout0, del_dout1, del_dout2, del_dout3, pannul_dout0, pannul_dout1, pannul_dout2, pannul_dout3, align32_dout0, align32_dout1, align32_dout2, align32_dout3);

ibuffer ib_unit (

clock, rst_buf, bin_stall, inst_dout0, inst_dout1, inst_dout2, inst_dout3, val_dout0, val_dout1, val_dout2, val_dout3, del_dout0, del_dout1, del_dout2, del_dout3, pannul_dout0, pannul_dout1, pannul_dout2, pannul_dout3, align32_dout0, align32_dout1, align32_dout2, align32_dout3, issued0, issued1, issued2, issued2, issued3, flushed0, inst_bout0, inst_bout1, inst_bout2, inst_bout3, val_bout0, val_bout1, val_bout2, val_bout3, del_bout0, del_bout1, del_bout2, del_bout3, pannul_bout0, pannul_bout1, pannul_bout2, pannul_bout3, align32_bout1, align32_bout2, align32_bout3, stall_bc_b);

group g_unit (

clock, rst_grp, issue_stall, reg_asi, inst_bout0, inst_bout1, inst_bout2, inst_bout3, val_bout0, val_bout1, val_bout2, val_bout3, del_bout0, del_bout1, del_bout2, del_bout3, pannul_bout0, pannul_bout1, pannul_bout2, pannul_bout3, align32_bout1, align32_bout2, align32_bout3, flush_ads, e_stall, pre_anuld_e, ieu0_out, ieu1_out, cti_out, mem_out, fa_out, fm_out, multi_out, fm_cmp_pixr_e, fm_cmp_pixr_c, fm_cmp_pixr_n1, ieu0_e, ieu1_e, call_e, call_c, jmpl_e, jmpl_c, return_e, return_c, bicc_e, bicc_c, bicc_n1, bpcc_e, bpcc_c, bpcc_n1, bpr_e, bpr_c, bpr_n1, fbfcc_e, fbfcc_c, fbfcc_n1, fbpfcc_e, fbpfcc_c, fbpfcc_n1, annulb_e, annulb_c, annulb_n1, ld_blk_two, ldfsr_b5adr, load_b3an3, load_ndr, load_b3adr, load_9out, store_8out, store_out, stfsr_e, stfsr_c, storef_e, st_blk_out, fm_divsqrt_3bf, movcc_e, movr_e, loadi_e, loadf_e, fa_addsub_e, fa_addsub_c, fa_addsub_n1, fa_convert_e, fa_convert_c, fa_convert_n1, fa_cmp_e, fa_cmp_c, fa_cmp_n1, fa_cmpcc_e, fm_mul_e, fm_mul_c, fm_mul_n1, fm_divsqrts_b12, fm_divsqrts_b13, fm_divsqrtd_b22, fm_divsqrtd_b23, fa_man_e, fa_man_c,

```
fa_mov_e, fa_mov_c, fa_mov_n1, fa_pasrame_e, fa_pasrame_c, fa_pasrame_n1, fm_pack_e,
fm_pack_c, fm_pack_n1, fm_pmuldist_e, fm_pmuldist_c, fm_pmuldist_n1, fa_ds_e, fa_ds_c,
fa_ds_n1, fm_ds_e, fm_ds_c, fm_ds_n1, fa_cds_e, fa_cds_c, fa_cds_n1, fa_daddr_e, fa_daddr_c,
fa_daddr_n1, fm_daddr_e, fm_daddr_c, fm_daddr_n1, fm_daddr_ds, i1_daddr_e, i1_daddr_c,
save_e, restore_e, l0_ndr, l1_ndr, l2_ndr, l3_ndr, l4_ndr, l5_ndr, l6_ndr, l7_ndr, l8_ndr, l0_int,
l1_int, l2_int, l3_int, l4_int, l5_int, l6_int, l7_int, l8_int, l0_fp, l1_fp, l2_fp, l3_fp, l4_fp, l5_fp,
l6_fp, l7_fp, l8_fp, ld_daddr_e, l0_daddr, l1_daddr, l2_daddr, l3_daddr, l4_daddr, l5_daddr,
l6_daddr, l7_daddr, l8_daddr, issued0, flushed0, ill_excep0_g, unimfpop0_g, tnd_dctic0,
pre_anu_d0, slot0_ieu0, slot0_ieu1, slot0_cti, slot0_mem, slot0_fa, slot0_fm, seticc0_g, a_field0,
i_field0, x_field0, p_field0, fprnd_sbit0, alt_field0, mmask0, cmask0, b0_cc, t0_cc, m0_cc,
opf0_cc, c0_cc, b0_cond, m0_cond, b0_rcond, m0_rcond, rs1_addr0, rs2_addr0, rd_addr0,
asi_a0, issued1, pre_anu_d1, slot1_ieu0, slot1_ieu1, slot1_cti, slot1_mem, slot1_fa, slot1_fm,
seticc1_g, a_field1, i_field1, x_field1, p_field1, fprnd_sbit1, alt_field1, mmask1, cmask1, b1_cc,
c1_cc, b1_cond, b1_rcond, rs1_addr1, rs2_addr1, rd_addr1, asi_a1, issued2, pre_anu_d2,
slot2_ieu0, slot2_ieu1, slot2_cti, slot2_mem, slot2_fa, slot2_fm, seticc2_g, a_field2, i_field2,
x_field2, p_field2, fprnd_sbit2, alt_field2, mmask2, cmask2, b2_cc, c2_cc, b2_cond, b2_rcond,
rs1_addr2, rs2_addr2, rd_addr2, asi_a2, issued3, pre_anu_d3, slot3_cti, slot3_fa, slot3_fm,
a_field3, i_field3, p_field3, fprnd_sbit3, b3_cc, c3_cc, b3_cond, rs1_addr3, rs2_addr3,
rd_addr3);

endmodule
```

Abstract

An Optimized Instruction Issue Architecture of a 64-bit 4-way Superscalar Microprocessor

Byung In Moon
Department of Electronic Engineering
The Graduate School
Yonsei University

This dissertation presents design and verification of an issue unit which stores and issues instructions in a 4-way superscalar RISC(Reduced Instruction Set Computer) microprocessor. This microprocessor uses SPARC version 9 as the instruction set architecture. By adding VIS(Visual Instruction Set) for graphics functionality, the microprocessor supports graphics application programs efficiently. The issue unit receives instructions from a prefetch unit, and issues them in order at a rate of as high as four instructions in one cycle for maximum utilization of functional units. By using an instruction buffer, the unit decouples instruction fetch and issue to improve instruction issue rate.

The issue unit is composed of an instruction aligner, an instruction buffer and a grouping logic. The instruction aligner aligns four valid instructions, and the first valid instruction becomes the first output to the instruction buffer. The instruction

buffer aligns and stores instructions from the instruction aligner, and sends the earliest four available instructions to the grouping logic. The grouping logic decodes instructions dispatched from the instruction buffer, and issues them if they are free from data dependency and necessary functional units and register file ports are available.

The issue unit is described with behavioral and structural level HDL(Hardware Description Language). This HDL model allows logic synthesis for low level design. The result of simulation using C program shows average IPC of 1.8 on condition of 100% hit rate of instruction and data caches and 100% branch prediction accuracy. It is expected that instruction scheduling by an optimized compiler will make this unit have higher performance.

Key Words : 4-way superscalar, RISC microprocessor, issue unit,
instruction issue rate, instruction aligner, instruction buffer,
grouping logic