6장. PCI

PCI(Peripheral Component Interconnect, 주변장치 상호연결)는 이름 그대로, 시스템에 있는 여러 주변장치들을 어떻게 구조적이고 관리를 잘 할 수 있는 방식으로 함께 연결할 것인지 를 정의하고 있는 표준이다. 이 표준[3, PCI 로컬버스 규약]은 시스템 장치들을 전기적으로 연결하는 방식과, 각 장치들이 동작해야 하는 방식을 규정하고 있다. 이 장에서는 리눅스 커널이 시스템의 PCI 버스들과 장치들을 초기화하는 방법을 간단히 살펴보도록 한다[1].

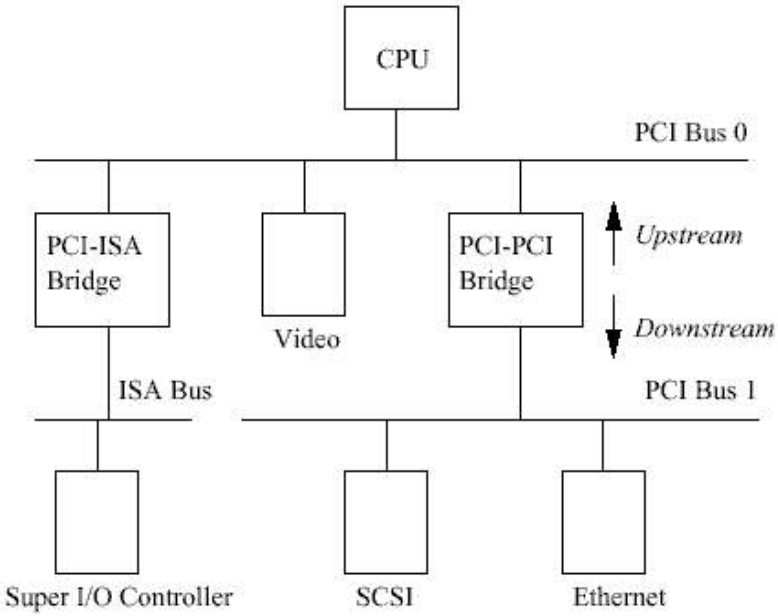

그림 6.1 : PCI 기반 시스템의 예

그림 6.1은 일반적인 PCI 기반 시스템의 논리 구성도이다. PCI 버스와 PCI-PCI 브릿지는 시 스템 장치들을 서로 연결하는 접착제와 같은 것이다. CPU는 첫번째 PCI 버스인 0번 PCI 버 스에 연결되어 있고, 이 구성도에서는 비디오 장치가 여기에 연결되어 있다. PCI-PCI 브릿지 는 특별한 PCI 장치로서, 1차(primary) PCI 버스와 2차(secondary) PCI 버스인 1번 PCI 버스를 연결한다[2]. PCI 규약에 나오는 전문용어로는, 1번 PCI 버스를 PCI-PCI 브릿지의 다운스트림 (downstream), 0번 버스를 브릿지의 업스트림(upstream)이라고 한다[3]. 이 구성도에서 2차 PCI 버스에는 SCSI 카드나 이더넷 카드 등이 연결된다. 물리적으로 브릿지와 2차 PCI 버스, 여 기 연결된 두 장치들은 하나의 복합 PCI 카드로 만들 수 있다[4]. PCI-ISA 브릿지는 오래전부 터 사용되어온 ISA 장치들을 지원하는 것으로, 이 구성도에서는 키보드와 마우스, 플로피 드라이브를 제어하는 슈퍼 I/O 콘트롤러 칩이 여기에 연결되어 있다.

6.1 PCI 주소공간(PCI Address Space)

CPU와 PCI 장치들은 그들이 공유하고 있는 메모리에 접근할 필요가 있다. 이 메모리는 디 바이스 드라이버가 PCI 카드를 제어하고 서로 정보를 교환하는데에 사용되는 것이다. 일반 적으로 이 공유 메모리에는 장치에 속한 제어 레지스터(control register)와 상태 레지스터 (status register)가 들어 있다. 이 레지스터들은 장치를 제어하고 상태를 읽는데 쓰인다. 예를 들어, PCI SCSI 디바이스 드라이버는 SCSI 디스크로 데이터를 쓰려고 할 때, 장치의 상태 레 지스터를 읽어서 장치가 쓸 준비가 되었는지 알아 내고, 이 값이 켜져 있을 때에 장치가 원 하는 동작을 하도록 제어 레지스터에 값을 쓰게 된다.

CPU가 관리하는 시스템 메모리를 이런 공유 메모리로 사용될 수도 있겠지만, 이렇게 한다 면 PCI 장치가 메모리에 접근할 때마다 CPU가 메모리에 접근하지 못한 채 PCI 장치가 작 업을 끝마치기를 기다려야 하는 문제가 발생할 것이다. 이는 일반적으로 메모리에 접근하는 것은 동시에 하나로 제한되어 있기 때문이며, 이렇게 한다면 시스템이 느려질 것이다. 그렇 다고 주변장치가 메인 메모리에 아무런 통제 없이 접근할 수 있도록 하는 것 역시 좋지 않 은 생각이다. 이것은 매우 위험하며, 잘못 만들어진 장치는 시스템을 매우 불안하게 만들 수 있다.

주변장치들은 각자 자신만의 메모리 공간을 가지고 있다. CPU는 이 영역에 자유롭게 접근할 수 있지만, 반대로 이 장치가 시스템 메모리에 접근하는 것은 DMA(Direct Memory Access, 직 접 메모리 접근) 채널을 이용하는 경우로만 엄격히 제한되어 있다. ISA 장치는 ISA I/O와 ISA 메모리라는 두가지 주소공간을 가진다. PCI는 세가지 주소공간을 가지는데, PCI I/O, PCI 메모리, 그리고 PCI 설정공간(configuration space)이 그것이다. CPU는 이들 주소공간 모두에 접근할 수 있는데, 디바이스 드라이버는 PCI I/O와 PCI 메모리 주소공간을 사용하며, PCI 설정공간은 리눅스 커널의 PCI 초기화 코드에서 사용하고 있다[5].

알파 AXP 프로세서는 원래 시스템 주소공간을 제외한 다른 주소공간에 접근할 수 없도록 되어 있다. 그래서 여기서는 PCI 설정 주소공간과 같은 다른 주소공간에 접근할 수 있도록 해주는 여러 칩셋을 사용한다. 그리고 거대한 가상 주소공간에서 일부를 빼내 PCI 주소공간 으로 맵핑하는 희소 주소 매핑(sparse address mapping)방법을 사용한다.

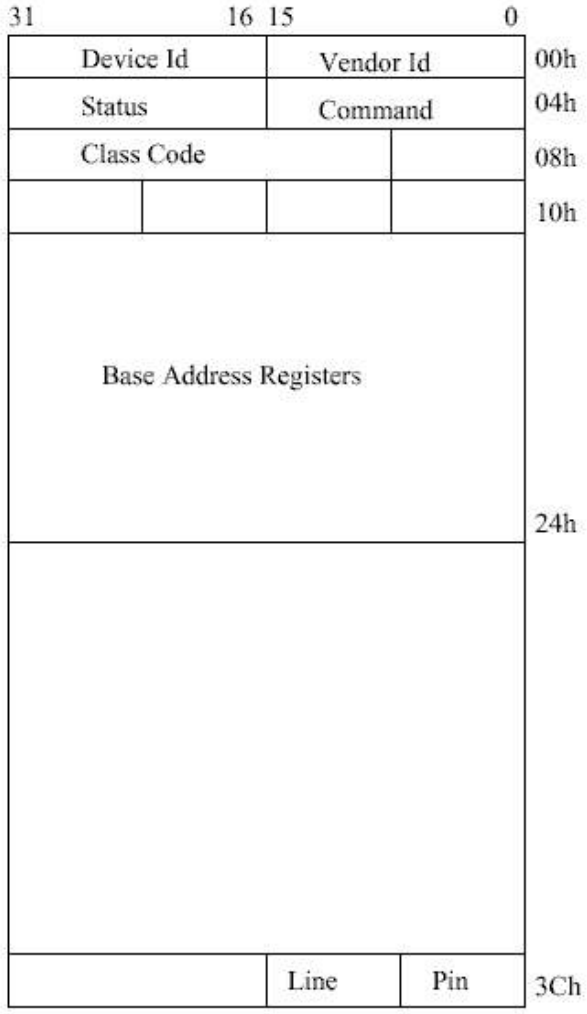

그림 6.2 : PCI 설정 헤더

6.2 PCI 설정 헤더(PCI Configuration Header)

모든 PCI 장치는 (PCI-PCI 브릿지도 포함하여) PCI 설정 주소공간 어딘가에 설정에 관련된 자료구조를 가지고 있다. PCI 설정 헤더는 시스템이 장치를 구별하고 제어할 수 있게 한다. 이 헤더가 있는 정확한 위치는 PCI 배치도에서 장치가 위치한 곳에 따라 결정된다. 예를 들 어, PCI 비디오 카드를 PC 메인보드에 있는 여러 PCI 슬롯 중 하나에 꽂을 때, 어떤 슬롯에 꽂느냐에 따라 PCI 설정 주소공간에서 각기 다른 위치에 헤더가 위치하게 된다. 이것은 그 다지 문제가 되지 않는데, 왜냐면 PCI 장치와 브릿지가 어디 있든간에, 시스템은 설정 헤더 에 있는 상태 레지스터, 설정 레지스터를 이용해 그들을 찾아내 설정할 것이기 때문이다.

보통 PCI 설정 헤더의 오프셋는 보드에서의 슬롯 번호에 관련이 있다. 그래서, 첫번째 슬롯 의 PCI 설정 헤더가 0번 오프셋에 위치 한다면, 두번째 슬롯의 헤더는 256번 오프셋에 위치 하고 (모든 헤더는 똑같이 256바이트 크기이다), 다른 슬롯의 헤더도 이런 식으로 위치하게 된다. 시스템별로 PCI 설정 영역에 접근하는 하드웨어 메커니즘이 다르게 정의되어 있으며, 이를 이용하여 PCI 설정을 하는 코드는 주어진 PCI 버스에 있어서 가능한 모든 PCI 설정 헤더를 검사하여, 어떤 장치가 있고 어떤 장치가 없는지를 헤더의 한 항목을 (보통 제작자 식별자 (Vendor Identification)[6] 항목) 읽고 에러를 검출함으로써 파악한다. [3, PCI 로컬 버스 규 약]에서는 빈 PCI slot 의 제작자 식별자나 장치 식별자(Device Identification)를 읽으려고 하면 0xFFFFFFFF을 돌려주는 것으로 에러 검출방법을 정의하고 있다.

그림 6.2는 256 바이트의 PCI 설정 헤더의 배치도를 보여준다. 여기에는 다음과 같은 항목 이 있다.

- **제작자 식별자(Vendor Identification)** PCI 장치의 제작자를 나타내는 고유번호. 예를 들어 디 지털(Digital)은 0x1011, 인텔은 0x8086를 고유번호로 갖는다.
- **장치 식별자(Device Identification)** 장치 자체를 나타내는 고유번호. 예를 들어 디지털의 21141 고속 이더넷 장치는 0x0009 값을 갖는다.
- **상태(Status)** 이 항목은 장치의 상태를 나타내는데, 각각의 비트들이 갖는 의미는 표준에서 정의하고 있다. [3, PCI 로컬 버스 규약]
- **명령(Command)** 시스템은 이 항목에 값을 씀으로써, 장치의 PCI I/O 메모리를 접근을 허가 하는 것 같은, 장치를 제어하는 일을 한다.
- **분류코드(Class Code)** 이 장치가 속한 장치의 유형을 구별한다. 모든 종류의 장치에 대해 비 디오, SCSI 같은 식의 표준 분류가 있다. SCSI에 대한 분류코드는 0x0100이다.
- **베이스 주소 레지스터(Base Address Register)** 이 레지스터는 장치가 사용하는 PCI I/O, PCI 메 모리 공간의 유형과 크기, 위치를 지정하는데 사용된다.
- **인터럽트 핀(Interrupt Pin)** PCI 카드에 있는 물리적인 핀 중 네 개는, 카드로부터 PCI 버스 로 인터럽트를 전달하는 역할을 한다. 표준에서는 이들을 각각 A, B, C, D라고 부른다[7]. 인터럽트 핀 항목은 PCI 장치가 이들 핀 중 어떤 핀을 사용하고 있는지 나타낸다. 보통 특정 장치에 있어서 인터럽트 핀은 직접 배선되어 있다. 즉, 시스템이 부팅할 때마다 그 장치는 똑같은 인터럽트 핀을 사용한다는 것이다. 이 정보는 인터럽트 처리 시스템이 장 치에서 발생하는 인터럽트를 관리할 수 있도록 해준다.
- **인터럽트 라인(Interrupt Line)** 이 항목은 PCI 초기화 코드와 디바이스 드라이버, 리눅스의 인터럽트 처리 서브시스템 사이에 인터럽트 핸들을 전달하기 위해 사용한다. 여기 있는 값은 디바이스 드라이버에겐 의미가 없겠지만, 인터럽트 핸들러가 PCI 장치로부터 온 인 터럽트를 리눅스 운영체제에 있는 올바른 디바이스 드라이버의 인터럽트 처리 코드로 인 터럽트를 전달할 수 있게 한다. 리눅스가 인터럽트를 처리하는 방법에 대해 자세한 것은 7장을 참조하라.

6.3 PCI I/O와 PCI 메모리 주소

이들 두 주소공간은 장치가 CPU 상의 리눅스 커널에서 실행되는 디바이스 드라이버와 통신 하기 위해 사용하는 곳이다. 예를 들어, DECchip 21141 고속 이더넷 장치는 자신의 내부 레 지스터를 PCI I/O 공간에 매핑한다. 그러면 해당하는 리눅스 디바이스 드라이버는 장치를 제 어하기 위해서 이들 레지스터를 읽고 쓴다. 비디오 드라이버는 비디오 정보를 저장하기 위 해 방대한 양의 PCI 메모리 공간을 사용한다.

이들은 PCI 시스템이 셋업이 되고, PCI 설정 헤더에 있는 명령(Command) 항목에서 이들 주 소공간에 대한 접근을 허용할 때까지, 아무도 이 공간에 접근할 수 없다. 여기서 PCI를 설 정하는 코드만이 PCI 설정 영역을 읽고 쓸 수 있으며, 리눅스 디바이스 드라이버는 단지 PCI I/O와 PCI 메모리 공간만 읽고 쓸 수 있다는 사실을 기억하기 바란다.

6.4 PCI-ISA 브릿지(Bridge)

이 브릿지는 PCI I/O, PCI 메모리 공간으로의 접근을 ISA I/O, ISA 메모리 공간으로의 접근으 로 바꾸어 줌으로써, 오래전부터 사용해온 ISA 장치를 지원하는 역할을 한다. 지금 팔리는 많은 시스템들은 PCI 버스 슬롯과 함

께 여러개의 ISA 버스 슬롯을 가지고 있는데, 시간이 지날수록 이전 것과 호환성을 유지할 필요는 줄어들고, 언젠가는 PCI 슬롯만 있는 시스템이 팔릴 것이다[8]. ISA 주소공간(I/O와 메모리 공간)에서 ISA 장치의 레지스터가 위치해 있는 곳 은, 안개가 자욱한 시절에 나온 초창기 8080 기반 PC에 의해 고정되었다. 5000 달러가 넘는 알파 AXP 기반 컴퓨터조차도 ISA 플로피 콘트롤러는 처음 나온 IBM PC에서와 똑같은 I/O 공간을 사용한다[9]. PCI 규약에서는 이 문제를 PCI I/O와 메모리 주소공간에서 아래쪽 영역을 ISA 시스템의 ISA 주변장치 용으로 예약을 하고, 하나의 PCI-ISA 브릿지를 통해 PCI 메모 리로의 접근을 이 영역으로 바꾸어 줌으로써 해결한다.

6.5 PCI-PCI 브릿지

PCI-PCI 브릿지는 시스템에 있는 PCI 버스들을 붙여주는 특별한 PCI 장치이다. 간단한 시스 템에는 PCI 버스가 하나밖에 없지만, 하나의 PCI 버스가 지원할 수 있는 PCI 장치의 개수에 는 전기적인 제한이 있어서, 더 많은 PCI 장치를 지원하기 위해서 PCI-PCI 브릿지를 통해 PCI 버스를 추가할 수 있도록 한다. 이는 특히 고성능 서버에 있어서 중요하다. 당연히, 리 눅스는 PCI-PCI 브릿지를 지원한다.

6.5.1 PCI-PCI 브릿지 : PCI I/O와 PCI 메모리 윈도우(Memory Window)

PCI-PCI 브릿지는 다운스트림으로 가는 PCI I/O나 PCI 메모리에 읽거나 쓰는 요청 중의 일 부만들 통과시킨다. 예를 들어 그림 6.1에서, 0번 PCI 버스에서 1번 PCI 버스로 가는 읽고 쓰는 명령이 있을 때, PCI-PCI 브릿지는 그 주소가 SCSI 카드나 이더넷 카드의 메모리 일 때에만 통과시켜주고, 그 외의 주소일 때는 무시해버린다. 이런 필터링은 필요없는 주소가 시스템 전체로 전달되는 것을 막아준다. 이를 위해 PCI-PCI 브릿지가 1차 버스(primary bus) 에서 2차 버스(secondary bus)로 전달해야 하는 PCI I/O와 PCI 메모리 공간의 베이스 주소와 범위를 프로그램해야 한다. 한번 PCI-PCI 브릿지를 설정하고 나면, 디바이스 드라이버가 이 윈도우를 통해서 PCI I/O와 PCI 메모리 공간에 접근하는 한, PCI-PCI 브릿지는 보이지 않는 다. 이는 PCI 디바이스 드라이버 제작자들을 편하게 해주는 중요한 특징이다. 나중에 살펴 볼 것이지만, 이는 리눅스가 PCI-PCI 브릿지를 설정하는 것을 좀 까다롭게 한다.

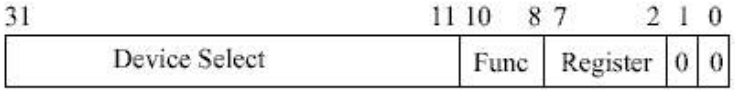

그림 6.3 : 0번 타입 설정 사이클

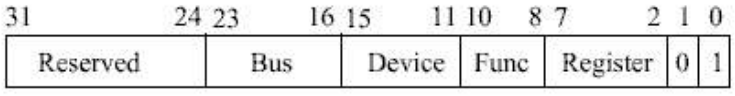

그림 6.4 : 1번 타입 설정 사이클

6.5.2 PCI-PCI 브릿지 : PCI 설정 사이클(Configuration Cycle)과 PCI 버스에 번호 붙이기

PCI 초기화 코드가 메인 PCI 버스(0번 PCI 버스)에 있지 않는 장치들에 접근할 수 있으려면, 브릿지가 자신의 1차 인터페이스에서 2차 인터페이스로 설정 사이클[10]를 넘길건지 말건지를 결정하도록 할 수 있는 메커니즘이 있어야 한다. 사이클이란 PCI 버스에서 보이는 주소이다. PCI 규약에서는 두가지 형식의 PCI 설정 주소를 정의

하고 있다. 이는 0번 타입과 1번 타입 인데 그림 6.3과 6.4에서 비교해서 보여주고 있다. 0번 타입 PCI 설정 사이클에는 버스 번호 가 없고, 모든 장치는 이를 같은 PCI 버스에 대한 PCI 설정 주소로 해석한다. 0번 타입 설정 사이클에서 11-31번 비트는 장치 선택 항목이다. 시스템을 디자인하는 방법중 하나는 각각 다른 장치에 다른 비트를 부여하는 것인데, 여기서는 비트 11은 0번 슬롯에 있는 PCI 장치 를, 비트 12는 1번 슬롯에 있는 PCI 장치를 선택한다. 다른 방법은 장치의 슬롯 번호를 바 로 11-31 비트에 써 넣는 것이다. 이 중 어떤 방법을 사용하는지는 시스템의 PCI 메모리 컨 트롤러에 따라 다르다.

1번 타입 PCI 설정 사이클에는 PCI 버스 번호가 있으며, 이 타입의 설정 사이클은 PCI-PCI 브릿지만 사용하고 다른 장치들은 모두 무시한다. 모든 PCI-PCI 브릿지는 1번 타입 설정 사 이클을 보았을 때, 그것을 자신의 PCI 버스의 다운스트림으로 통과시킬지를 결정할 수 있다. PCI-PCI 브릿지가 1번 타입 설정 사이클을 무시할건지, 아니면 다운스트림 PCI 버스로 통과 시킬 것인지는, PCI-PCI가 어떻게 설정되었느냐에 달려있다. 모든 PCI-PCI 브릿지는 1차 버 스 인터페이스 번호와 2차 버스 인터페이스 번호를 가지고 있다. 1차 버스 인터페이스는 CPU에 더 가까운 쪽에 있는 버스이고, 2차 버스 인터페이스는 더 멀리 있는 것이다. 또한 각 PCI-PCI 브릿지는 종속 버스(subordinate bus) 번호를 가지고 있는데, 이는 2차 버스 인터 페이스 너머 브릿지로 연결된 모든 PCI 버스에서 최대 버스 번호이다. 다르게 표현하면, 종 속 버스 번호는 PCI-PCI 브릿지의 다운스트림으로 연결된 PCI 버스 번호 중 가장 큰 값이다. PCI-PCI 브릿지가 1번 타입 설정 사이클을 만나면, 브릿지는 다음 중에서 한가지 일을 한다.

- 지정된 버스 번호가 브릿지의 2차 버스 번호와 종속 버스 번호 사이에 있지 않으면 (이 두 버스 번호도 포함하여) 무시한다.
- 버스 번호가 브릿지의 2차 버스 번호와 일치하면 0번 타입 설정 명령으로 변환한다.
- 지정된 버스 번호가 2차 버스 번호보다 크고 종속 버스 번호보다 작거나 같으면, 바 꾸지 않고 그대로 2차 버스 인터페이스로 통과시킨다.

따라서 그림 6.9에 있는 배치도에서 3번 버스에 있는 장치1을 지정하고자 한다면, CPU로부 터 1번 타입 설정 명령을 만들어야 한다. 그러면 브릿지1은 이를 바꾸지 않고 1번 버스로 통과시키고, 브릿지2는 이를 무시하며, 브릿지3은 이를 0번 타입 설정 명령으로 바꾸고 3번 버스로 보내 장치1이 응답하게 된다.

PCI 설정을 할 때 버스에 번호를 할당하는 것은 개별 운영체제에 따라 다르겠지만, 어떤 방 식으로 번호를 붙이든간에 다음 명제는 시스템에 있는 모든 PCI-PCI 브릿지에 있어서 참이 어야 한다.

"PCI-PCI 브릿지 너머에 있는 모든 PCI 버스의 번호는, 2차 버스 번호와 종속 버스 번호 (이 들을 포함하여) 사이에 있어야 한다."

만약 이 규칙이 깨진다면, PCI-PCI 브릿지는 1번 타입 설정 사이클을 통과시키지 않거나 제 대로 변환하지 못할 것이고, 시스템은 자신에 있는 PCI 장치를 찾지 못하거나 초기화하지 못할 것이다. 리눅스는 번호를 올바로 붙이기 위해서 이들 특별한 장치들(PCI-PCI 브릿지)을 특정한 순서로 설정한다. 리눅스에서 PCI 브릿지와 버스에 번호를 붙이는 방식은 6.6.2장에 서 실제 예제와 함께 설명한다.

6.6 리눅스 PCI 초기화

리눅스에서 PCI 초기화 코드는 다음 세가지 논리적인 부분으로 쪼갤 수 있다.

- **PCI 디바이스 드라이버** 이 유사 디바이스 드라이버(pseudo device driver)[11]는 0번 버스부터 PCI 시스템을 찾기 시작하여, 시스템에 있는 모든 PCI 장치와 브릿지들을 찾는다. 그리 고 해당 자료구조의 연결 리스트를 만들어 시스템의 배치도를 나타낸다. 더불어, 찾은 모 든 브릿지에 번호를 부여한다.

- **PCI BIOS** 이 소프트웨어 계층은 [4, PCI BIOS ROM 규약]에 기술된 서비스들을 제공한다. 알 파 AXP에서는 이런 BIOS 서비스가 없지만, 똑같은 일을 하는 동등한 코드가 리눅스 커 널에 포함되어 있다.

- **PCI 확정(PCI Fixup)** 시스템별로 다른 이 코드는 시스템별로 다른 PCI 초기화의 잡다한 종 료부분을 깨끗하게 마무리한다.

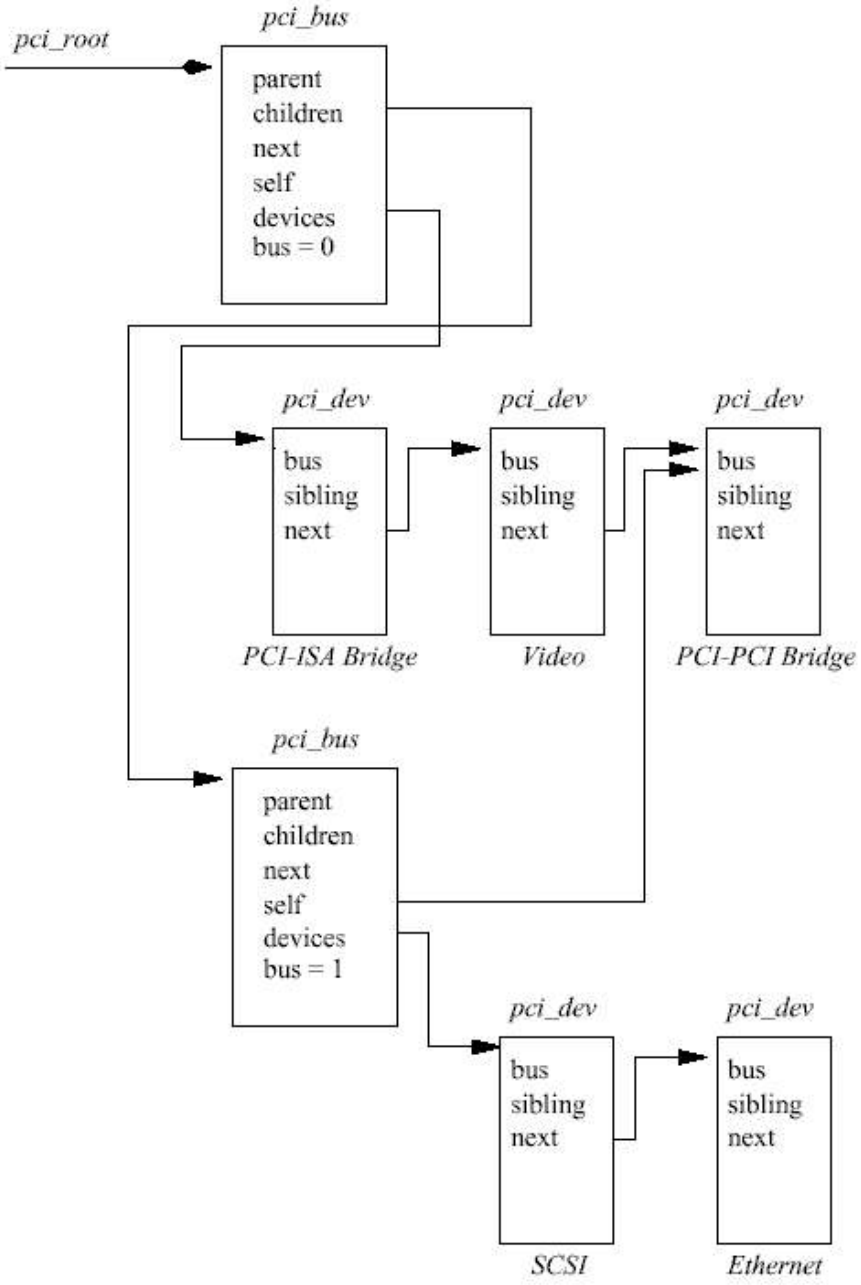

그림 6.5 : 리눅스 커널 PCI 자료구조

6.6.1 리눅스 커널 PCI 자료구조

리눅스 커널은 PCI 시스템을 초기화하면서 시스템의 실제 PCI 배치도를 그대로 나타내는 자료구조를 만든다. 그림 6.5는 그림 6.1에서 예로 든 PCI 시스템에 대해 만들어지는 자료구 조를 보여준다.

각 PCI 장치는 (PCI-PCI 브릿지도 포함하여) pci_dev 자료구조로, PCI 버스는 pci_bus 자 료구조로 나타낸다. 최종결과는 버스들의 트리 구조로 트리의 각 노드는 자신에 연결된 여 러 PCI 장치들을 자식으로 가진다. PCI버스는 PCI-PCI 브릿지를 통해서만 도달할 수 있으므 로 (첫번째 PCI 버스인 0번 버스를 제외하고), 각 pci_bus 자료

구조는 접근할 때 거쳐야 하는 PCI-PCI 브릿지에 대한 포인터를 갖는다. 이 PCI 장치는 PCI 버스의 부모 PCI 버스의 자식이다.

그림 6.5에는 나오지 않지만 시스템에 있는 모든 PCI 장치에 대한 포인터인 pci_devices 가 있다. 시스템에 있는 모든 PCI 장치는 자신의 pci_dev 자료구조를 가지고 있고, 이들은 이 큐(pci_devices)에 들어있다. 이 큐는 리눅스 커널이 시스템에 있는 모든 PCI 장치를 빨리 찾는데 사용한다.

6.6.2 PCI 디바이스 드라이버

PCI 디바이스 드라이버는 실제 디바이스 드라이버가 아니라, 시스템 초기화 때 불리는 운영 체제의 한 함수이다. PCI 초기화 코드는 시스템에 있는 모든 PCI 버스에서 모든 PCI 장치들 (PCI-PCI 브릿지를 포함하여)을 조사해야 한다. 이 코드는 PCI BIOS 코드를 이용하여, 현재 조사하고 있는 PCI 버스의 모든 가능한 슬롯이 점유되어 있는지 아닌지 확인한다. 그리고 그 PCI 슬롯이 점유되어 있으면, 그 장치를 기술하는 pci_dev 자료구조를 만들고, 존재하 는 PCI 장치의 리스트(pci_devices에서 가리키고 있다)에 이를 추가한다

PCI 초기화 코드는 0번 PCI 버스부터 찾기 시작한다. 이 코드는 가능한 모든 PCI 슬롯에서 가능한 모든 PCI 장치의 제작자 식별자(Vendor Identification)와 장치 식별자(Device Identification)를 읽으려고 한다. 그리고 점유되어 있는 슬롯을 발견하면, 그 장치를 나타내는 pci_dev 자료구조를 만든다. PCI 초기화 코드에 의해 만들어진 모든 pci_dev 자료 구조는 PCI-PCI 브릿지를 포함하여 모두 pci_devices라는 단일 연결 리스트에 연결된다.

만약 찾은 PCI 장치가 PCI-PCI 브릿지라면, pci_bus 자료구조를 만들어 pci_bus 트리와 pci_root가 가리키고 있는 pci_dev 자료구조에 연결한다. PCI 초기화 코드는 장치의 분 류 코드가 0x060400 라는 것으로 그 PCI 장치가 PCI-PCI 브릿지임을 알 수 있다. 그리고 나 서 리눅스 커널은 자신이 찾은 PCI-PCI 브릿지의 다른 쪽(다운스트림)의 PCI 버스를 설정한 다. 또 다른 PCI-PCI 브릿지를 발견하더라도 똑같은 방법으로 설정한다. 이 과정은 깊이탐색 (depthwise) 알고리즘이라고 하는 방법이다. 시스템의 PCI 배치도는 넓이탐색(breadthwise)을 하기 전에 깊이탐색을 통하여 완전히 구성이 된다. 그림 6.1을 보면 리눅스가 0번 PCI 버스 에 있는 비디오 장치를 설정하기 전에, 1번 PCI 버스에 있는 이더넷과 SCSI 장치를 설정했 음을 알 수 있다.

리눅스가 다운스트림 PCI 버스를 찾을 때, 중간에 있는 PCI 브릿지의 2차 버스 번호와 종속 버스 번호를 설정해야 하는데, 이는 다음에 자세하게 설명하고 있다.

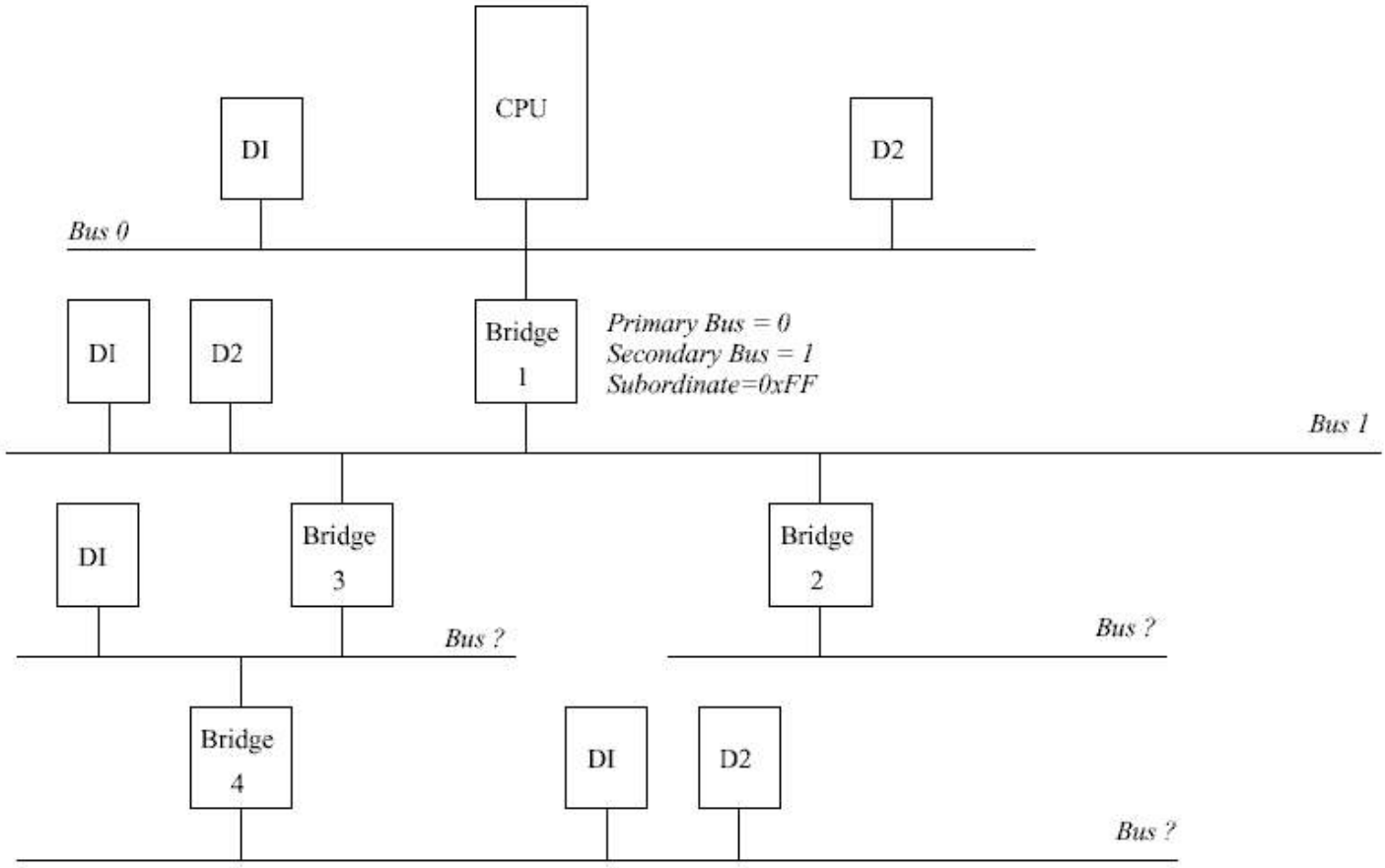

그림 6.6 : PCI 시스템 설정 : 1단계

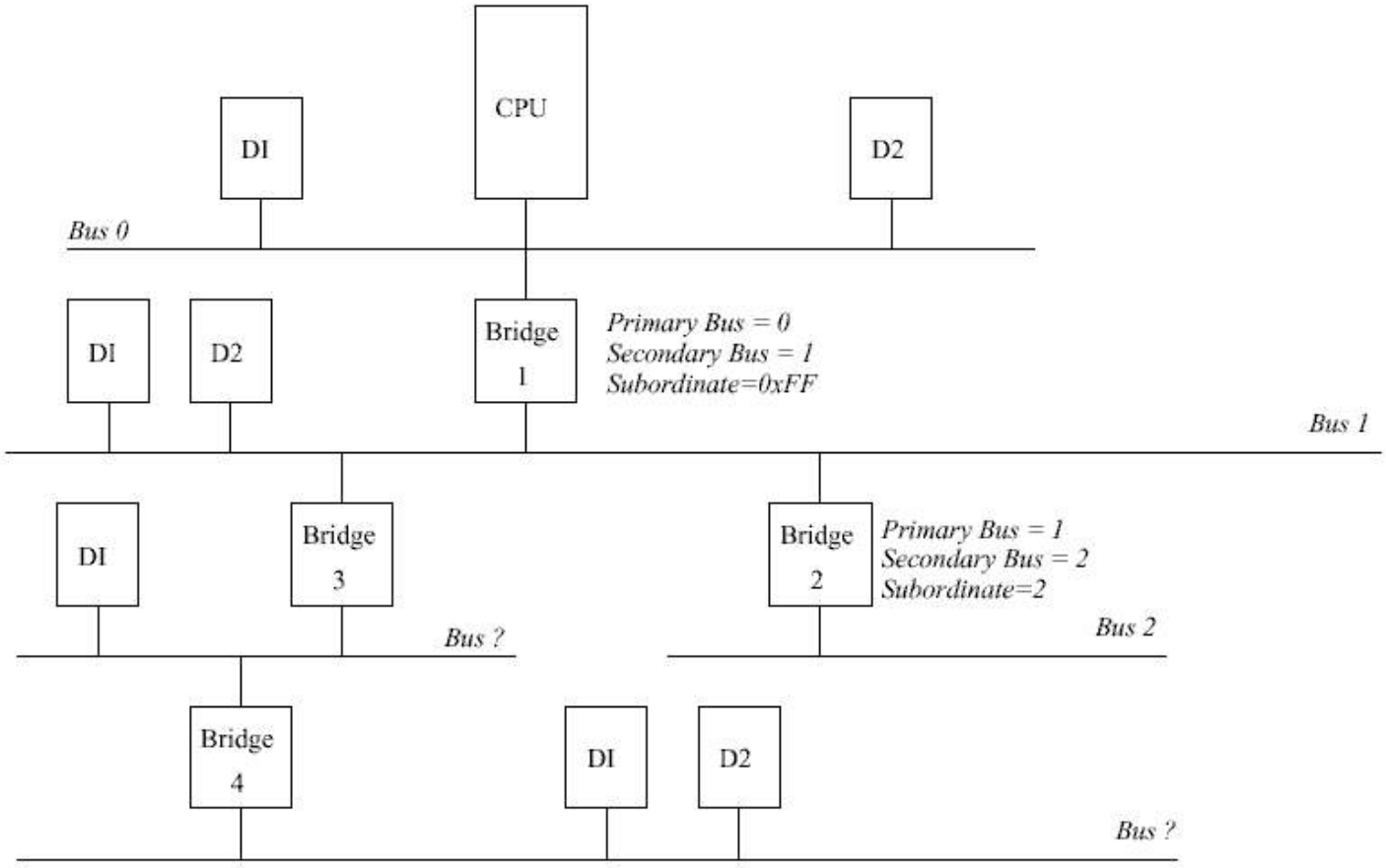

그림 6.7 : PCI 시스템 설정 : 2단계

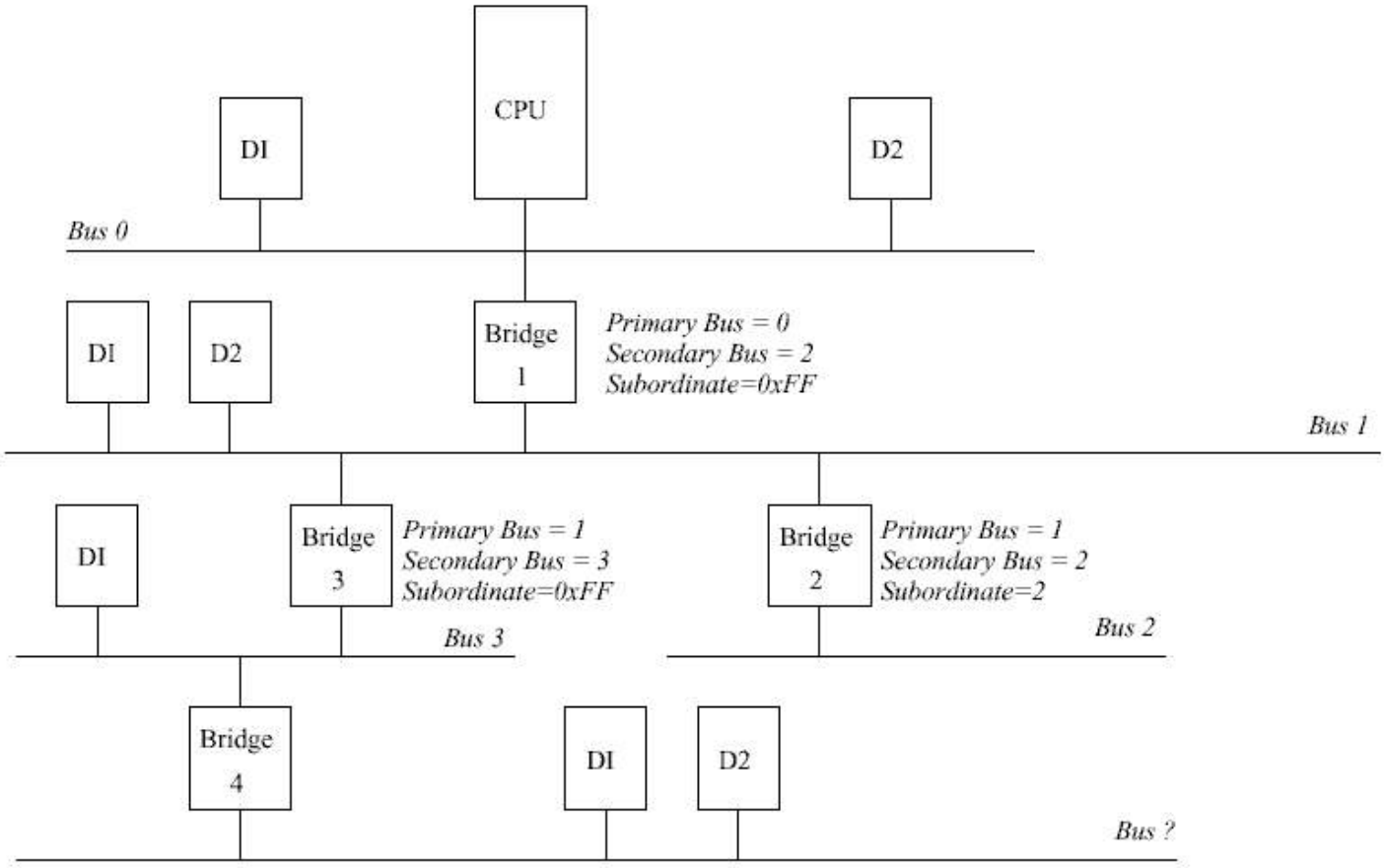

그림 6.8 : PCI 시스템 설정 : 3단계

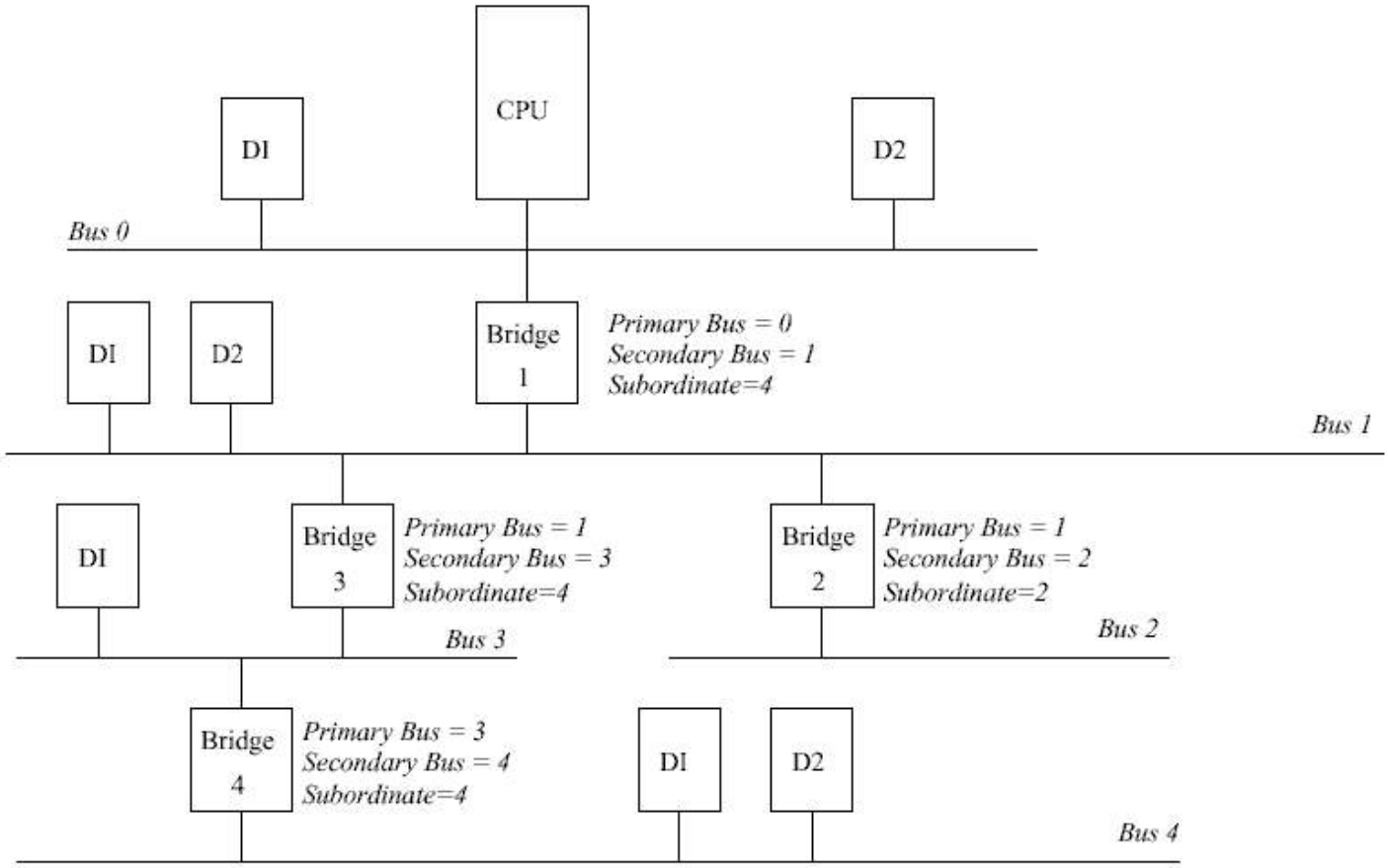

그림 6.9 : PCI 시스템 설정 : 4단계

PCI-PCI 브릿지 설정하기 - PCI 버스 번호 부여하기 PCI-PCI 브릿지가, PCI I/O, PCI 메모리, 또는 PCI 설정 주소 공간에 읽고 쓰려는 시도를 통과 시킬 수 있으려면, 브릿지는 다음과 같은 사항을 알아야 한다 :

- **1차 버스(primary bus)** 번호 PCI-PCI 브릿지에 바로 연결된 업스트림 버스 번호
- **2차 버스(secondary bus)** 번호 PCI-PCI 브릿지에 바로 연결된 다운스트림 버스 번호
- **종속 버스(subordinate bus)** 번호 브릿지의 다운스트림으로 도달할 수 있는 버스들의 가장 큰 번호
- **PCI I/O와 PCI 메모리 윈도우** PCI-PCI 브릿지의 모든 다운스트림에서 사용하는 PCI I/O 주 소공간과 PCI 메모리 주소공간의 윈도우에 대한 베이스 주소와 크기

문제는 어떤 주어진 PCI-PCI 브릿지를 설정하려고 할 때 그 브릿지의 종속 버스 번호를 알 수 없다는 것이다. 다운스트림으로 PCI-PCI 브릿지가 더 있는지 모르고, 안다고 하더라도 그 것이 어떤 번호를 갖게 될지도 모른다. 해답은 깊이탐색 재귀 알고리즘을 사용하여, 각 버스 에 있는 PCI-PCI 브릿지를 조사하고, 찾으면 번호를 부여하는 것이다. PCI-PCI 브릿지를 찾 으면, 2차 버스에 번호를 붙이고, 임시적으로 종속 버스 번호에 0xFF를 지정한 후, 다운스트 림으로 PCI-PCI 브릿지를 찾아 계속 번호를 붙여나간다. 이것은 복잡해 보이겠지만, 아래에 있는 실제 동작하는 예제를 보면 이 과정이 더 명쾌해질 것이다.

- **PCI-PCI 브릿지 번호붙이기 : 1 단계** 그림 6.6의 배치도에서 처음 찾게 되는 브릿지는 브릿 지1이다. 브릿지1의 다운스트림 PCI 버스는 1번이 되며, 브릿지1의 2차 버스 번호로 1번 이, 그리고 임시적으로 종속 버스 번호로 0xFF가 할당된다. 이는 PCI 버스 번호로 1번이 나 이 이상을 지정한 1번 타입 설정 사이클은 모두 브릿지1을 통과하여 PCI 버스 1번으 로 가게된다는 것이다. 만약 1번 타입 설정 사이클의 버스 번호가 1번이라면 이는 0번 타입 설정 사이클로 변환이 되겠지만, 다른 버스 번호라면 변환되지 않고 그대로 있

을 것이다. 이 과정은 리눅스의 PCI 초기화 코드가 1번 PCI 버스로 진행하여 조사하기 위해 해야 하는 것이다.

- **PCI-PCI 브릿지 번호붙이기 : 2단계** 리눅스는 깊이탐색 알고리즘을 사용하므로, 초기화 코 드는 1번 PCI 버스로 진행하여 이를 조사한다. 여기서 PCI-PCI 브릿지2를 발견하게 되고, PCI-PCI 브릿지2를 넘어서는 더이상 PCI-PCI 브릿지가 없으므로 종속 버스 번호와 2차 인터페이스 번호로 똑같이 2를 갖게 된다. 그림 6.7은 이 시점에서 버스와 PCI-PCI 브릿 지가 어떤 값을 갖게 되는지 보여준다.
- **PCI-PCI 브릿지 번호붙이기 : 3단계** PCI 초기화 코드는 1번 PCI 버스를 조사하는 곳으로 되 돌아와 다른 PCI-PCI 브릿지인 브릿지3을 찾게 된다. 이는 1차 버스 번호로 1을, 2차 버 스 번호로 3을, 그리고 종속 버스 번호로 0xFF를 갖게 된다. 그림 6.8에는 이 때 시스템 이 어떻게 설정되는지 보여준다. 이제 버스 번호로 1, 또는 2나 3이 지정된 1번 타입 PCI 설정 사이클은 해당하는 PCI 버스로 올바르게 배달될 것이다.
- **PCI-PCI 브릿지 번호붙이기 : 4단계** 리눅스는 브릿지3의 다운스트림인 3번 버스를 조사하기 시작한다. 3번 PCI 버스는 다른 PCI-PCI 브릿지(브릿지4)를 갖고 있고, 이는 1차 버스 번 호로 3을, 2차 버스 번호로 4를 부여받는다. 이것은 이 줄기에서 가장 끝에 있는 브릿지 이므로, 종속 버스 번호로 4를 부여받는다. 초기화 코드는 PCI-PCI 브릿지3으로 돌아와 종속 버스 번호로 4를 지정한다. 마지막으로 PCI 초기화 코드는 PCI-PCI 브릿지1의 종속 버스 번호로 4를 할당할 수 있게 된다. 그림 6.9는 최종적인 버스 번호를 보여준다.

6.6.3 PCI BIOS 함수

PCI BIOS 함수들은 모든 플랫폼들에서 공통적인 표준 함수들 시리즈 중의 하나이다. 예를 들어, 이들은 인텔 기반 시스템과 알파 AXP 기반 시스템에 있어 동일하다. 이들은 CPU가 모든 PCI 주소공간에 제어를 위해 접근할 수 있게 한다. 리눅스 커널 코드와 디바이스 드라 이버만이 이를 사용할 수 있다.

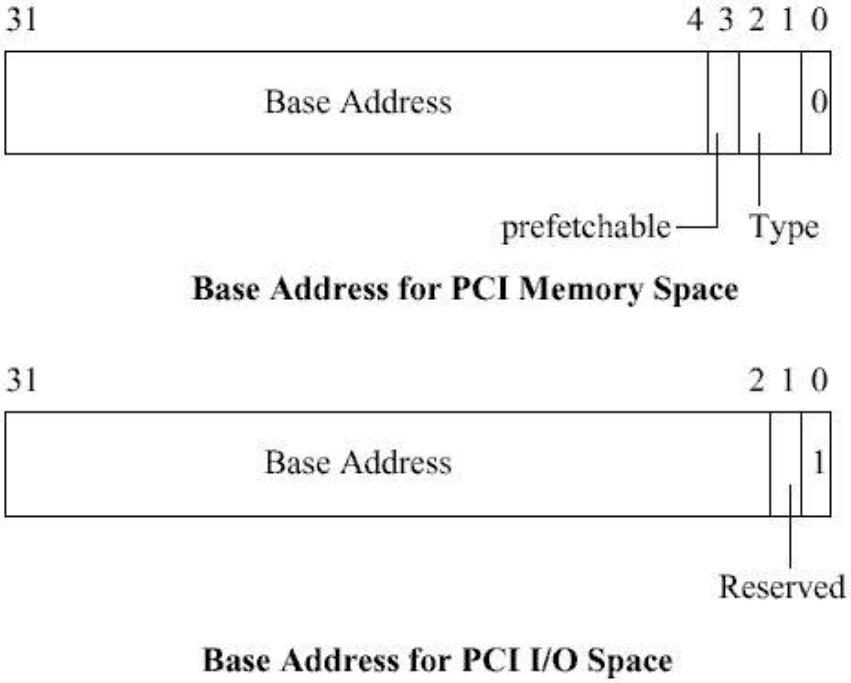

그림 6.10 : PCI 설정 헤더 : 베이스 주소 레지스터들

6.6.4 PCI 확정(PCI Fixup)

알파 AXP용 PCI 확정 코드는 인텔용(기본적으로 아무것도 하지 않는 코드이다)보다 훨씬 많다. 인텔 기반 시스템은 부팅시에 실행되는 시스템 BIOS를 가지고 있으며, 이것이 PCI 시 스템의 설정을 이미 끝내버렸기 때문이다. 그래서 리눅스는 이미 설정되어 있는 것을 매핑 하는 것 외에는 할 일이 거의 없다. 그러나 인텔 기반이 아닌 시스템에서는 다음과 같은 설 정이 필요하다[12].

- 각 장치에 PCI I/O와 PCI 메모리 공간을 할당한다.
- 시스템에 있는 각 PCI-PCI 브릿지의 PCI I/O와 PCI 메모리 주소 윈도우를 설정한다.
- 장치에 인터럽트 라인 값을 만든다. 이것은 그 장치의 인터럽트 처리를 제어한다.

다음 작은 절에서는 이들 코드가 어떻게 동작하는지 이야기한다.

장치가 얼마나 많은 PCI I/O와 PCI 메모리 공간을 필요로 하는지 알아내기

찾은 PCI 장치들이 얼마나 많은 PCI I/O와 메모리 주소공간을 필요로 하는지 알아내기 위해 서 이를 각 장치에게 물어야 한다. 이는 각 장치의 베이스 주소 레지스터에 전부 1을 써넣 고 나서 이를 다시 읽어봄으로써 이루어진다. 장치는 자신에게 무관한 주소 비트를 0으로 설정하고, 이는 자신이 필요로 하는 주소공간의 크기를 나타내는 효과를 가지게 된다.

베이스 주소 레지스터에는 두가지 기본 유형이 있는데, 먼저 어떤 주소공간에 장치의 레지 스터가 있어야 하는지를 - PCI I/O 공간인지 PCI 메모리 공간인지 - 지시한다. 이것은 레지스 터의 비트 0으로서 알 수 있다. 그림 6.10에서는 PCI 메모리와 PCI I/O를 위한 베이스 주소 레지스터의 두가지 유형을 보여준다.

주어진 베이스 주소 레지스터가 얼마나 많은 주소공간을 요구하는지 알아내기 위해, 레지스 터에 모두 1을 써넣고 이를 다시 읽는다. 그러면 장치는 자신에게 무관한 주소 비트를 0으 로 설정하여, 필요한 주소공간의 크기를 지정하게 된다. 이런 디자인은 모든 주소공간의 크 기는 2의 배수이고 이에 맞춰 자연스럽게 정렬되어 있다는 것을 알려준다.

예를 들어 DECChip 21142 PCI 이더넷 장치를 초기화할 때 장치는, PCI I/O나 PCI 메모리로 0x100 바이트의 공간을 필요로 한다고 알려준다. 초기화 코드는 그만큼의 공간을 할당하고, 이 순간 21142의 제어 레지스터와 상태 레지스터를 이 주소에서 볼 수 있게 된다.

PCI-PCI 브릿지와 PCI 장치에 PCI I/O와 PCI 메모리를 할당하기

다른 메모리처럼 PCI I/O와 PCI 메모리 공간도 유한하며, 어느정도 희소한 것이다. 인텔기반 이 아닌 시스템에서의 PCI 확정 코드는 (그리고 인텔 시스템에서의 BIOS 코드는) 각 장치가 요구하는 크기의 메모리를 효과적인 방법으로 장치에게 할당해야 한다. PCI I/O와 PCI 메모 리는 자연스럽게 정렬이 되도록 장치에 할당되어야 한다. 예를 들어, 장치가 PCI I/O 공간 0xB0을 요구한다면 이것은 0xB0의 여러배가 되는 주소에서 정렬되어야 한다는 것이다. 여기 에 더해, 어떤 브릿지든지 PCI I/O와 PCI 메모리 베이스는 4K 단위로 정렬되어야 하며 각자 1MByte의 경계를 가지고 있어야 한다. 다운스트림 장치들의 주소공간이 모든 업스트림 PCI- PCI 브릿지의 메모리 공간에 위치해 있어야 하기 때문에, 공간을 효율적으로 할당하는 것은 조금 어려운 문제이다.

리눅스는 PCI 디바이스 드라이버가 만들어낸 버스/장치 트리에서 기술된 각 장치들을 주소 공간에서 PCI I/O 메모리가 증가하는 쪽으로 할당하는 알고리즘을 사용한다. 여기서도 재귀 적인 알고리즘을 사용하여 PCI 초기화 코드에서 만들어낸 pci_bus와 pci_dev 자료구조를 따라간다. PCI 버스의 루트부터(pci_root가 가리키고 있는) 시작하여 BIOS 확정 코드는 다 음과 같이 하게 된다.

- 현재 있는 전역 PCI I/O와 메모리 베이스를 4K 단위로 상대적으로 1Mbyte 경계를 가지도록 정렬한다.
- 현재 버스의 모든 장치에 대해 (증가하는 쪽으로 PCI I/O 메모리를 필요로 한다)

- 장치에게 PCI I/O와 PCI 메모리 공간을 할당하고
- 행다하는 크기만큼 전역 PCI I/O와 메모리 베이스를 이동하고
- 장치의 PCI I/O와 PCI 메모리를 사용하는 것을 가능하게 한다.

- 현재 버스의 모든 다운스트림 버스들을 재귀적으로 찾아 공간을 할당한다. 이는 전 역 PCI I/O와 메모리 베이스를 바꾼다는 것에 주의한다.
- 현재 있는 전역 PCI I/O와 PCI 메모리 베이스를 4K 단위로 상대적으로 1Mbyte 경계 로 정렬하고, 이렇게 하는 중에 현재 PCI-PCI 브릿지가 필요로 하는 PCI I/O와 PCI 메모리 윈도우의 베이스와 크기를 알아낸다.
- 이 버스에 연결된 PCI-PCI 브릿지를 프로그램하여 PCI I/O와 PCI 메모리 베이스와 크기를 지정한다.
- PCI-PCI 브릿지에 있는 PCI I/O와 PCI 메모리에 접근하도록 하는 브릿지 기능을 켠 다. 이는 브릿지의 1차 PCI 버스에서 보이는 PCI I/O와 PCI 메모리 주소 중에서 윈 도우 안에 있는 PCI I/O와 PCI 메모리 주소는 2차 버스로 건너가게 된다는 것이다.

예로 들었던 그림 6.1을 생각하면, PCI 확정 코드는 다음과 같이 시스템을 설정할 것이다 : PCI 베이스들의 정렬 (Align the PCI bases) PCI I/O는 0x4000, PCI 메모리는 0x100000이다. 이것 은 PCI-ISA 브릿지가 모든 주소를 ISA 주소 사이클로 변환할 수 있게 한다.

비디오 장치 이 장치는 0x200000만큼의 PCI 메모리를 필요로 하여, 현재 PCI 메모리 베이스 0x200000에서 시작하는 크기를 할당해주는데, 요구한 크기대로 자연히 정렬이 되어야 한 다. PCI 메모리 베이스는 0x400000으로 이동하고, PCI I/O 베이스는 그대로 0x4000에 남아 있다.

PCI-PCI 브릿지 이제 PCI-PCI 브릿지를 건너가 거기에서 PCI 메모리를 할당한다. 여기서 베이스들이 이미 올바로 정렬이 되어 있기 때문에 정렬을 할 필요가 없다.

- **이더넷 장치** 이것은 0xB0 바이트의 PCI I/O와 PCI 메모리 공간을 요구한다. 이 장치는 0x4000에서 시작하는 PCI I/O 공간과 0x400000에서 시작하는 PCI 메모리를 갖게 된 다. PCI 메모리 베이스는 0x4000B0으로 이동하고 PCI I/O 베이스는 0x40B0이 된다.

- **SCSI 장치** 이것은 0x1000 크기의 PCI 메모리를 요구하고, 자연 정렬이 된 후 0x401000 에서 시작하는 메모리를 할당받는다. PCI I/O 베이스는 그대로 0x40B0, PCI 메모리 베 이스는 0x402000으로 이동한다.

PCI-PCI 브릿지의 PCI I/O, PCI 메모리 윈도우 이제 브릿지로 돌아와 0x4000과 0x40B0 사이 에 위치하는 PCI I/O 윈도우와, 0x400000과 0x402000 사이에 위치하는 PCI 메모리 윈도우 를 설정한다 이것은 PCI-PCI 브릿지가 비디오 장치에 대한 접근은 무시하고, 이더넷이나 SCSI 장치로 접근할 때에만 이를 통과시키도록 한다.

번역 : 이호
정리 : 이호

역주 1) 이 장의 내용과 분량은 일반적으로 운영체제에 대해 가지는 관심에 비해 자세하고 많은 편이다. 이는 PCI BIOS가 모든 일을 처리하는 인텔 기반 PC와는 달리, 다른 시스템 에서는 PCI 버스를 운영체제가 직접 제어를 해야하며, 저자가 주로 알파 AXP 기반 시스 템에서 작업을 했기 때문인 것 같다. 하지만 버스 구조에 대한 이해는 실제 하드웨어와 관련된 작업에 있어서 큰 도움을 주리라 생각한다. (flyduck)

역주 2) 하나의 PCI 버스가 감당할 수 있는 장치의 갯수가 한정되어 있기 때문에, 더 많은 장치를 연결하려면 버스를 추가하고 이를 PCI-PCI 브릿지로 연결해야 한다. 예전에는 일 반 PC에는 PCI-PCI 브릿지가 없고, 서버용 기계에만 PCI-PCI 브릿지가 있었지만, 요즘에 는 일반 PC 용으로도 PCI-PCI 브릿지가 있는 보드가 나오고 있다. MS Windows 운영체제 를 사용하고 있다면 시스템 등록정보의 장치관리자에서 시스템 장치에 어떤 것이 연결되 어 있는지 확인해보면 좋을 것이다. CPU 버스와 PCI 버스를 연결하는 브릿지와, PCI-ISA 브릿지, PCI-PCI 브릿지 등을 확인할 수 있을 것이다. (flyduck)

역주 3) CPU의 입장에서 생각한다면 0번 버스가 CPU에 바로 연결된 것이기 때문에, 1번 버 스에서 0번 버스로 가는 것은 위로 가는 것이므로 업스트림, 0번 버스에서 1번 버스로 가 는 것은 다운스트림이 된다. (flyduck)

역주 4) PCI 규약에 따라 하나의 PCI 카드가 최대 8개의 기능을 가질수 있다. (flyduck)

역주 5) 처음 IBM PC가 나올 때부터 여기에는 메모리 공간외에 I/O 공간에 있는 포트라는 것이 있었다. 이는 보통의 메모리 접근 명령이 아닌 특수한 포트 입출력 명령을 사용했 는데 인텔 CPU에 있는 in, out 명령이 그것이다. ISA 카드에서는 장치의 레지스터에 접근 하는데 이런 포트 I/O를 사용하였고, 큰 영역의 데이터에 접근할 때는(예를 들어 그래픽 카드의 메모리) 메모리 영역을 사용하였다. 하지만 인텔 CPU와는 달리 많은 CPU들은 포트 I/O를 지원하지 않는다. 비록 PCI 규약이 I/O 공간을 지원하기 하지만, 이를 사용하지 않는 PCI 카드들도 있으며, 여기서는 칩의 레지스터들이 모두 메모리 공간에 존재한다. ISA 카드에서는 64KB의 I/O 영역과 640KB-1MB, 15MB-16MB(이 영역을 사용하는 장치는 아주 드물다)의 메모리 영역을 사용하고 있다. 이는 지금까지 유효한데, 몇가지 문제를 야기하고 있다. PCI에서는 4GB의 I/O 공간과, 32비트 또는 64비트의 메모리 공간을 제공한다. (flyduck)

역주 6) 여기서 제작자는, PCI 카드의 제작자라기 보다는 카드에 있는 PCI와 연결 역할을 하 는 칩의 제작자이다. (flyduck)

역주 7) 이 이름은 IRQ A-D와 무관하며, PCI 카드상의 핀에 대한 이름이다. (flyduck)

역주 8) PCI가 등장한 초창기에는 빠른 입출력 속도를 필요로 하는 장치들만 PCI 카드로 나 왔지만, PCI의 여러 장점으로 인해 지금은 거의 모든 카드가 PCI 용으로 제작되고 있다. ISA 카드는 제작자의 입장에서 만들기 쉽다는 장점이 있지만, 사용자의 입장에서는 설정 하기가 까다롭다는 단점 때문에 갈수록 찾아보기 힘들어지고 있다. (flyduck)

역주 9) ISA 장치의 가장 큰 단점은 사용하는 I/O 공간과 메모리 공간, IRQ 등이 하드웨어적 으로 고정되어 있다는 점이다. 몇 장치들은 관습처럼 고정되어 그대로 이어져오고 있으 며, 다른 장치들은 하드웨어에 있는 점퍼로 설정하거나, EPROM에 설정값을 기록해놓고 있다. (flyduck)

역주 10) 설정 사이클이란 PCI가 초기화가 되지 않았을 때 PCI 장치들을 설정하기 위하여 사용하는 특별한 주소를 말한다. (flyduck)

역주 11) 실제로 디바이스 드라이버인 것이 아니라, 디바이스 드라이버처럼 장치를 구동하 는 역할은 하지만 디바이스 드라이버 형태를 갖추지 않은 것이므로 유사 디바이스 드라 이버라고 한다. 이런 것으로는 가상 파일 시스템이나 네트웍 드라이버가 있다. (flyduck)

역주 12) IBM PC외에 BIOS가 있는 시스템을 찾기 힘들며, 이런 시스템에서는 운영체제 코드 가 그 역할을 맡아야 한다. (flyduck)