사례
명령어 집합 컴퓨터 감소

데이비드 A. 패터슨

컴퓨터 과학과 캘 리 포 니

아 대학교 버클리, 캘리포

니아 94720

데이비드 R. 디첼

벨 연구소 컴퓨팅 과학 연구

센터

머레이 힐, 뉴저지 07974

소개

컴퓨터 아키텍트의 주요 목표 중 하나는 이전 컴퓨터보다 비용 효율적인 컴퓨터를 설계하는 것입니다. 비용 효율성에는 컴 퓨 터 제조를 위한 하드웨어 비용, 프로그래밍 비용, 초기 하드웨어와 후속 프로그램을 디버깅할 때 아키텍처와 관련하여 발생하는 비용이 모두 포함됩니다. 컴퓨터 제품군의 역사를 살펴보면 가장 일반적인 아키텍처 변화는 점점 더 복잡한 기계로 변화하는 추세라는 것을 알 수 있습니다. 아마도 이러한 추가적인 복잡성은 최신 모델의 비용 효율성과 관련하여 긍정적인 측면이 있을 것입니다. 이 백서에서는 이러한 추세가 항상 비용 효율적인 것은 아니며, 오히려 득 보 다 실이 많을 수도 있다고 제안합니다. 축소 명령어 집합 컴퓨터**(RISC)**가 복합 명령어 집합 컴퓨터**(CISC)**만큼 비용 효율적일 수 있는 경우를 살펴보겠습니다. 이 백서에서는 차세대 VLSI 컴퓨터가 CISC보다 RISC로 더 효과적으로 구현될 수 있다고 주장할 것입니다.

이러한 복잡성 증가의 예로 **IBM** System/3에서 System/38[Utley78]로의 전환과 DEC PDP-11에서 VAX11로의 전환을 생각해 보십시오. 복잡성은 제어 저장소의 크기로 정량적으로 표시되며, DEC의 경우 PDP 11/40의 256 x 56에서 VAX 11/780의 5120 x 96으로 증가했습니다.

복잡성이 증가하는 이유

컴퓨터가 더 복잡해진 이유는 무엇일까요? 몇 가지 이유를 생각해 볼 수 있습니다:

메모리 속도 v. CPU의 속도. 존 코크는 701에서 709로 넘어가면서 복잡성이 시작되었다고 말합니다[Cocke80]. 701 CPU는 코어 메인 메모리보다 약 10배 빨랐기 때문에 서브루틴으로 구현된 프리미티브는 명령어인 프리미티브보다 훨씬 느렸습니다. 따라서 부동 소수점 서브루틴은 극적인 성능 향상과 함께 709 아키텍처의 일부가 되었습니다. 709를 더 복잡하게 만들면 701보다 비용 효율이 더 높아지는 발전이 이루어졌습니다. 그 이후로 성능 향상을 위해 기계에 많은 '상위 레벨' 명령어가 추가되었습니다. 이러한 경향은 속도의 불균형 때문에 시작되었지만, 설계자들이 이러한 불균형이 여전히 설계에 유효한지 스스로에게 물어본 적이 있는지 여부는 분명하지 않습니다.

마이크로코드 및 LSI 기술. 마이크로프로그래밍 제어를 사용하면 유선 제어보다 비용 효율적으로 콤플렉스 아키텍처를 구현할 수 있습니다[Husson70]. 60년대 후반과 70년대 초반에 이루어진 집적 회로 메모리의 발전으로 인해 거의 모든 경우에 마이크로프로그래밍된 제어가 더 비용 효율적인 접근 방식이 되었습니다. 마이크로프로그래밍 제어를 사용하기로 결정하면 명령어 세트를 확장하는 데 드는 비용은 매우 적으며, 단지 몇 단어의 제어 메모리를 더 저장하면 됩니다. 제어 메모리의 크기가 2의 거듭제곱인 경우가 많기 때문에 마이크로프로그래밍을 확장하여 제어 메모리를 완전히 채우면 추가 하드웨어 비용 없이 명령어 집합을 더 복잡하게 만들 수 있는 경우도 있습니다. 따라서 구현 기술의 발전으로 기존 서브루틴을 아키텍처로 옮기는 아키텍처를 비용 효율적으로 구현할 수 있게 되었습니다. 이러한 명령어의 예로는 문자열 편집, 정수에서 부동 소수점으로의 변환, 다항식 평가와 같은 수학적 연산이 있습니다.

코드 밀도. 초기 컴퓨터의 경우 메모리가 매우 비쌌습니다. 따라서 매우 컴팩트한 프로그램을 만드는 것이 비용 효율적이었습니다. 복잡한 명령어 집합은 종종 "상위" 코드 압축으로 예고됩니다. 그러나 명령어 집합의 복잡성을 높여 코드 밀도를 높이려는 시도는 명령어와 주소 지정 모드가 많아질수록 이를 표현하는 데 더 많은 비트가 필요하기 때문에 양날의 검이 될 수 있습니다. 증거에 따르면 코드 압축은 원래 명령어 집합을 정리하는 것만으로도 쉽게 달성할 수 있습니다. 코드 압축도 중요하지만, 10% 더 많은 메모리를 사용하는 비용이 아키텍처 "혁신"을 통해 CPU에서 10%를 짜내는 비용보다 훨씬 저렴한 경우가 많습니다. 대규모 CPU의 비용은 추가 회로 패키지가 필요한 반면, 단일 칩 CPU의 비용은 더 큰(따라서 더 느린) 제어 PLA로 인해 성능이 저하될 가능성이 더 높습니다.

마케팅 전략 안타깝게도 컴퓨터 회사의 주요 목표는 가장 비용 효율적인 컴퓨터를 설계하는 것이 아니라 컴퓨터를 판매하여 가장 많은 수익을 창출하는 것입니다. 컴퓨터를 판매하려면 제조업체는 자사 설계가 경쟁사 제품보다 우수하다는 것을 고객에게 설득해야 합니다. 복잡한 명령어 세트는 확실히 더 나은 컴퓨터의 주요 "표시" 증거입니다. 설계자는 일자리를 유지하기 위해 내부 경영진에게 새롭고 더 나은 설계를 계속 판매해야 합니다. 복잡한 명령어 집합의 실제 사용이나 비용 효율성에 관계없이 명령어의 수와 그 '파워'는 아키텍처를 홍보하는 데 자주 사용됩니다. 어떤 의미에서 컴퓨터 구매자가 복잡성 대 비용 효율성 문제에 대해 의문을 제기하지 않는 한 제조업체와 설계자는 이에 대해 비난받을 수 없습니다. 실리콘 하우스의 경우, 상대적으로 저렴한 CPU와 함께 대량의 메모리를 구매하도록 고객을 유인하는 것이 실질적인 이익이기 때문에 멋진 마이크로프로세서가 종종 드로우 카드로 사용됩니다.

상향 호환성. 마케팅 전략과 더불어 상향 호환성에 대한 필요성도 인식되고 있습니다. 상향 호환성이란 디자인을 개선하는 주된 방법이 새롭고 일반적으로 더 복잡한 기능을 추가하는 것임을 의미합니다. 아키

텍처에서 명령어 또는 주소 지정 모드가 제거되는 경우는 거의 없으며, 그 결과 일련의 컴퓨터에서 명령어의 수와 복잡성이 점진적으로 증가하게 됩니다. 새로운 아키텍처는 성공적인 경쟁사의 기계에서 발견되는 모든 명령어를 포함하는 경향이 있는데, 이는 아마도 아키텍트와 고객이 무엇이 "좋은" 명령어 집합을 정의하는지에 대해 제대로 파악하지 못 하 기 때문일 수 있습니다.

고급 언어 지원. 고급 언어의 사용이 점점 더 대중화됨에 따라 제조업체는 이를 지원하기 위해 더 강력한 명령어를 제공하고자 합니다. 안타깝게도 더 복잡한 명령어 집합이 실제로 그러한 지원을 제공했다는 증거는 거의 없습니다. 오히려 많은 경우 복합 명령어 집합이 유용하기보다는 오히려 해롭다고 주장할 수 있습니다. 고급 언어를 지원하려는 노력은 칭찬할 만하지만, 종종 잘못된 문제에 초점을 맞추고 있다고 생각합니다.

멀티프로그래밍 사용. 시간 공유가 증가함에 따라 컴퓨터는 실행 중인 프로세스를 중단하고 나중에 다 시 시작할 수 있 는 인터럽트 기능에 대응할 수 있어야 했습니다. 메모리 관리와 페이징은 명령이 완료되기 전에 중단했다가 나중에 다시 시작할 수 있는 기능도 추가로 요구했습니다. 이 중 어느 것도 명령어 세트 자체의 설계에 큰 영향을 미치지는 않았지만 구현에 직접적인 영향을 미쳤습니다. 복잡한 명령어와 주소 지정 모드는 인터럽트 시 저장해야 하는 상태를 증가시킵니다. 이 상태를 저장하려면 섀도 레지스터를 사용해야 하고 마이크로코드의 복잡성이 크게 증가합니다. 이러한 복잡성은 복잡한 명령어나 부작용이 있는 주소 지정 모드가 없는 머신에서는 대부분 사라집니다.

CISC는 어떻게 사용되어 왔나요?

소프트웨어 비용 상승의 흥미로운 결과 중 하나는 고급 언어에 대한 의존도가 높아진다는 것입니다. 그 결과 중 하나는 컴파일러 작성자가 기계가 실행할 명령어를 결정할 때 어셈블리 언어 프로그래머를 대체하고 있다는 것입니다. 컴파일러는 복잡한 명령어를 활용하지 못하는 경우가 많으며, 어셈블리 언어 프로그래머가 즐겨 사용하는 교묘한 트릭을 사용하지도 않습니다. 컴파일러와 어셈블리 언어 프로그래머는 또한 주어진 시공간 트레이드오프 하에서 유용하지 않은 명령어 집합의 일부를 당연히 무시합니다. 그 결과 아키텍처의 아주 작은 부분만 사용되는 경우가 많습니다.

예를 들어, 특정 IBM 360 컴파일러를 측정한 결과 10개의 명령어가 실행된 모든 명령어의 80%를 차지했고, ld는 90%, 21개는 95%, 30개는 99%를 차지한 것으로 나타났습니다[Alex- ander75]. 다양한 컴파일러와 어셈블리 언어 프로그램에 대한 또 다른 연구에서는 "CDC-3600의 명령어 집합을 현재 사용 가능한 명령어 중 R 또는 Ya로 줄이면 유연성이 거의 사라질 것"이라고 결론지었습니다."[Foster7l] Shustek은 IBM 370에 대해 "여러 번 관찰된 것처럼 프로그램 실행의 대부분을 차지하는 옵코드는 극소수에 불과하다"고 지적합니다. 예를 들어 COBOL 프로그램은 사용 가능한 183개의 명령어 중 84개를 실행하지만 48개는 실행된 모든 명령어의 99.08%를, 26개는 90.28%를 차지합니다."[Shustek78] 주소 지정 모드의 사용을 엑사마이닝할 때도 유사한 통계가 발견됩니다.

CISC 구현의 결과

기술의 급격한 변화와 CISC 구현의 어려움으로 인해 몇 가지 흥미로운 결과가 나 타 났 습 니 다 .

더 빠른 메모리. 반도체 메모리의 발전으로 CPU와 메인 메모리 간의 상대적인 속도 차이에 대한 가정에 몇 가지 변화가 생겼습니다. 반도체 메모리는 빠르고 상대적으로 저렴합니다. 최근 많은 시스템에서 캐시 메모리를 사용하면서 CPU와 메모리 속도 차이가 더욱 줄어들었습니다.

비합리적인 구현. 아마도 복합 아키텍처 구현의 가장 특이한 측면은 "합리적인" 구현이 어렵다

는 점일 것입니다. 이는 특수 목적의 명령어가 일련의 단순한 명령어보다 항상 빠른 것은 아니라는 의미입니다. 한 가지 예는 Peuto와 Shustek이 IBM 370에서 발견한 것입니다[Peuto,Shustek77]; 이들은 로드 명령어 시퀀스가 4개 미만의 레지스터에 대한 로드 다중 명령어보다 빠르다는 것을 발견했습니다. 이 사례는 일반적인 프로그램에서 로드 다중 명령어의 40Wr을 다룹니다. 다른 하나는 VAX-11/780에서 나왔습니다. INDEX 명령어는 배열 요소의 주소를 계산하는 동시에 인덱스가 배열 바운드에 맞는지 확인하는 데 사용됩니다. 이는 고급 언어 구문에서 오류를 정확하게 감지하는 데 중요한 기능임이 분명합니다. VAX 11/780의 경우, 이 단일 "상위 수준" 명령어를 몇 가지 간단한 명령어(COMPARE, 점프 덜 서명되지 않음, 추가, **곱하기)**로 대체하면 동일한 기능을 45% 더 잘 수행할 수 있다는 사실을 발견했습니다.

더 빨라졌습니다! 또한 컴파일러가 하한이 0인 경우를 활용하면 간단한 명령어 시퀀스가 60% 더 빨라졌습니다. 분명히 더 작은 코드가 항상 더 빠른 코드를 의미하는 것은 아니며, '더 높은 수준의' 명령어가 더 빠른 코드를 의미하는 것도 아닙니다.

설계 시간 연장. 간혹 간과되는 비용 중 하나는 새로운 아키텍처를 개발하는 데 걸리는 시간입니다. **CISC의** 복제 비용은 낮을지 몰라도 설계 시간은 크게 늘어납니다. DEC는 PDP-1을 설계하고 납품을 시작하는 데 6개월밖에 걸리지 않았지만, 이제 VAX와 같은 기계의 경우 동일한 주기를 거치는 데 최소 3년이 걸립니다. 이렇게 긴 설계 시간은 결과물인 구현의 품질에 큰 영향을 미칠 수 있습니다. 3년이나 지난 기술로 기계가 발표되거나 설계자가 좋은 구현 기술을 예측하고 기계를 제작하는 동안 그 기술을 개척하려고 노력해야 하기 때문입니다. 설계 시간이 단축되면 결과물인 기계에 매우 긍정적인 영향을 미칠 것은 분명합니다.

설계 오류 증가. 복잡한 명령어 세트의 주요 문제 중 하나는 설계 디버깅이며, 이는 일반적으로 마이크로프로그래밍 제어에서 오류를 제거하는 것을 의미합니다. 문서화하기는 어렵지만, 거의 모든 제품군이 읽기 전용 제어 저장소를 사용했기 때문에 IBM 360 제품군에서는 이러한 수정이 주요 문제였을 가능성이 높습니다. 370 라인은 변경 가능한 제어 저장소를 독점적으로 사용하는데, 이는 하드웨어 비용이 감소했기 때문일 수도 있지만 360에서 오류가 발생했던 나쁜 경험 때문일 가능성이 더 높습니다. 제어 저장소는 플로피 디스크에서 로드되므로 운영 체제와 유사하게 마이크로코드를 유지 관리할 수 있으며, 버그를 수정하고 업데이트된 버전의 마이크로코드가 포함된 새 플로피를 현장에 배포할 수 있습니다. VAX 11/780 설계 팀은 마이크로코드 오류가 발생할 가능성이 있음을 깨달았습니다. 이에 대한 해결책은 필드 프로그래머블 로직 어레이와 1024워드의 쓰기 가능한 제어 저장소(WCS)를 사용하여 마이크로코드 오류를 패치하는 것이었습니다. 다행히 DEC는 자신들의 경험에 대해 더 개방적이었기 때문에 50개 이상의 패치가 이루어졌다는 것을 알고 있습니다. 마지막 오류가 발견되었다고 믿는 사람은 거의 없습니다.[2]

RISC 및 VLSI

단일 칩 VLSI 컴퓨터의 설계는 멀티 칩 SSI 구현보다 CISC의 위와 같은 문제를 더욱 심각하게 만듭니다. 몇 가지 요인으로 인해 감소된 명령어 집합 컴퓨터가 합리적인 설계 대안으로 제시되고 있습니다.

구현 가능성. 전체 CPU 설계를 하나의 칩에 담을 수 있느냐에 따라 많은 것이 달라집니다. 복잡한 아키텍처는 덜 복잡한 아키텍처보다 주어진 기술에서 실현될 가능성이 적습니다. 이에 대한 좋은 예로 DEC의

VAX 시리즈 컴퓨팅을 들 수 있습니다. 하이엔드 모델은 인상적으로 보일 수 있지만, 아키텍처의 복잡성으로 인해 현재의 설계 규칙으로는 완전히 불가능하지는 않더라도 단일 칩에서 구현하기가 매우 어렵습니다. VLSI 기술이 개선되면 결국 단일 칩 버전이 실현 가능하겠지만, 덜 복잡하지만 동등한 기능을 갖춘 32비트 아키텍처가 실현된 후에야 가능합니다. 따라서 **RISC** 컴퓨터는 더 이른 시기에 실현될 수 있다는 이점이 있습니다.

어떤 사람들은 다른 설명을 제시했습니다. 소프트웨어, 우편, 원자력 발전소 등 모든 것이 더 오래 걸리는데 컴퓨터만 안 될 이유가 있을까요? 젊고 굶주린 회사가 기존 회사보다 시간이 덜 걸릴 것이라는 의견도 있었습니다. 이러한 관찰은 DEC의 경험을 부분적으로 설명할 수 있지만, 어떤 상황이든 아키텍처의 복잡성이 설계 주기에 영향을 미칠 것이라고 생각합니다.

[2] 각 패치는 여러 개의 마이크로 인스트럭션을 WCS에 넣어야 한다는 것을 의미하므로 50개의 패치에는 252개의 마이크로 인스트럭션이 필요합니다.

tions. 복잡한 VAX 명령어에서 오류가 발생할 가능성이 높았기 때문에 이 중 일부는 WCS에서만 구현되어 패치와 기존 명령어에서 1024 단어의 상당 부분을 사용합니다.

설계 시간. 설계 난이도는 VLSI 컴퓨터의 성공에 결정적인 요소입니다. VLSI 기술이 대략 2 년 마 다 칩 밀도가 두 배 이상 증가한다면, 설계와 디버깅에 2년만 걸 리 는 설계는 훨씬 우수한 기술을 사용할 수 있으므로 4년이 걸리는 설계보다 더 효과적일 수 있습니다. 새 마스크의 처리 시간은 일반적으로 몇 개월 단위로 측정되기 때문에 오류가 발생할 때마다 제품 배송이 한 분기씩 지연되며, 일반적인 예로 Z8000 및 MC68000의 경우 1~2년 지연이 있습니다.

속도. 비용 효율성에 대한 궁극적인 테스트는 구현이 주어진 알고리즘을 실행하는 속도입니다. 칩 면적을 더 잘 활용하고 디버깅 시간을 줄여 최신 기술을 사용할 수 있으면 칩의 속도가 빨라집니다. RISC는 더 단순한 설계만으로도 속도가 향상될 수 있습니다. 단일 주소 모드 또는 명령어를 사용하면 제어 구조가 덜 복잡해질 수 있습니다. 이는 다시 더 작은 제어 PLA, 더 작은 m.icrocode 메모리, 기계의 임계 경로에 있는 더 적은 수의 게이트로 이어질 수 있으며, 이 모든 것이 *. " 래스터 마이너 사이클 시간으로 이어질 수 있습니다. 명령어 또는 주소 모드를 제외하면 기계의 마이너 사이클 속도가 10% 빨라진다면, 추가적으로 기계의 속도를 10% 이상 높여야 비용 효율이 높아집니다. 지금까지 복잡한 명령어 세트가 이런 방식으로 비용 효율적이라는 확실한 증거는 거의 없었습니다.

칩 영역 활용도 향상. 면적이 있다면 CISC를 구현해 보는 건 어떨까요? 주어진 칩 면적에 대해 실현할 수 있는 것에는 많은 트레이드오프가 있습니다. 우리는 **CISC** 아키텍처가 아닌 **RISC** 아키텍처를 설계함으로써 다시 얻은 면적을 사용하여 RISC를 CISC보다 훨씬 더 매력적으로 만들 수 있다고 생각합니다. 예를 들어, 실리콘 면적을 온칩 캐시[패터슨, 세퀸80], 더 크고 빠른 트랜지스터 또는 파이프라이닝에 사용하면 전체 시스템 성능이 더 향상될 수 있다고 생각합니다. **VLSI** 기술이 개선됨에 따라 **RISC** 아키텍처는 항상 비슷한 **CISC보다** 한 발 앞서 나갈 수 있습니다. **CISC가** 단일 칩에서 구현 가능해지면 **RISC는 파이프라이닝** 기술을 사용할 수 있는 실리콘 면적을 확보하게 **되고, CISC가** 파이프라이닝이 가능해지면 **RISC는** 온칩 캐시 등을 확보할 수 있습니다. 또한 CISC는 본질적인 복잡성으로 인해 고급 기술을 구현하기가 더욱 어렵다는 단점이 있습니다.

고급 언어 컴퓨터 시스템 지원

어떤 사람들은 아키텍처를 단순화하는 것이 하이레벨 언어 지원에서 후 퇴 하 는 것이라고 주장할 수 있습니다. 최근 논문[Ditzel,Patterson80]에서는 '고수준' 아키텍처가 반드시 고수준 언어 컴퓨터 시스템을 구현하는 데 있어 가장 중요한 요소는 아니라고 지적합니다. 하이레벨 언어 컴퓨터 시스템은 다음과 같은 특징을 갖는 것으로 정의되었습니다:

(1) *모든 프로그래밍, 디버깅 및 기타 사용자/시스템 상호 작용에 상위 수준 언어를 사용합니다.*

(2) *고급 언어 소스 프로그램 측면에서 구문 및 실행 시간 오류를 발견하고 보고합니다.*

(3) *사용자 프로그래밍 언어에서 내부 언어로 변환되는 외형적인 외관이 없습니다.*

따라서 유일한 중요한 특징은 하드웨어와 소프트웨어의 조합을 통해 프로그래머가 항상 높은 수준의 언어로 컴퓨터와 상호 작용할 수 있다는 것입니다. 프로그래머는 프로그램을 작성하거나 디버깅할 때 하위 수준을 인식할 필요가 없습니다. 이 요구 사항만 충족되면 목표는 달성된 것입니다. 따라서 다음에는 차이가 없습니다.

사실, 그 반대의 증거도 있습니다. TI ASC의 수석 아키텍트인 하비 크래건은 이 머신이 루프 내부의 인덱싱된 참조 성능을 개선하기 위해 복잡한 메커니즘을 구현했다고 말했습니다. 이러한 연산을 더 빠르게 실행하는 데는 성공했지만, 다른 상황에서는 ASC가 느려진다고 느꼈습니다. 그 결과 Cray[Cragon 80]에서 설계한 더 단순한 컴퓨터보다 ASC가 느려졌습니다.

언어의 토큰을 일대일로 매핑하는 CISC로 구현하든, 매우 빠르지만 간단한 기계로 동일한 기능을 제공하든 상관없이 고급 언어 컴퓨터 시스템입니다.

컴파일러의 경험에 따르면 명령어 집합이 단순하고 균일할 때 컴파일러 작성자의 부담이 완화됩니다. 높은 수준의 함수를 지원한다고 알려진 복잡한 명령어는 컴파일러에서 생성할 수 없는 경우가 많습니다.4 복잡한 명령어는 명령어의 수준이 높아질수록 "잘못된" 함수를 구현할 가능성이 높아집니다. 함수가 너무 전문화되어 다른 연산에 쓸모없어지기 때문입니다.[5] 복잡한 명령어는 일반적으로 성능 손실이 거의 없거나 거의 없는 소수의 하위 명령어로 대체할 수 있습니다.[6] 복잡한 명령어에 대한 코드를 생성할 때 버그가 발생할 가능성이 높기 때문에 CISC용 컴파일러를 생성하는 데 걸리는 시간이 추가로 증가합니다.

컴파일러가 더 쉽게 작성할 수 있도록 설계된 복잡한 지침이 종종 목표를 달성하지 못한다는 증거는 상당히 많습니다. 여기에는 몇 가지 이유가 있습니다. 첫째, 명령어가 너무 많기 때문에 주어진 기본 연산을 수행하는 방법이 너무 많아 컴파일러와 컴파일러 작성자 모두에게 혼란스러운 상황입니다. 둘째, 많은 컴파일러 작성자는 실제로는 그렇지 않은데도 합리적인 구현을 다루고 있다고 가정합니다. 그 결과 "적절한" 명령어가 종종 잘못된 선택으로 판명되는 경우가 많습니다. 예를 들어, 스택의 레지스터를 **PUSHL R0으로** 푸시하는 것은 VAX 11/780에서 이동 명령어 MOVL R0,-(SP)로 푸시하는 것보다 느립니다. 이 외에도 거의 모든 복잡한 기계에 대해 수십 가지의 예가 더 있을 수 있습니다. "있기 때문에" 명령어를 사용하지 않도록 각별히 주의해야 합니다. 상대적인 명령어 타이밍을 변경하면 최적의 코드 생성을 유지하기 위해 새로운 코드 생성기가 필요하기 때문에 프로그램 이식성이나 훌륭한 컴파일러 작성자의 명성을 완전히 파괴하지 않고는 동일한 아키텍처의 다른 모델에서 이러한 문제를 "수정"할 수 없습니다.

상위 수준 언어를 지원하고자 하는 욕구는 HLLCS를 달성하는 것과 컴파일러 복잡성을 줄이는 것 모두를 포함합니다. **RISC가 CISC보다** 현저히 떨어지는 경우는 거의 **없으므로** 적절하게 설계된 **RISC는 CISC만큼이나** 하이레벨 언어를 지원하는 데 합리적인 아키텍처라는 결론을 내릴 수 있습니다.

찬성과 반대의 증거는 DEC에서 나옵니다. 서브루틴 호출의 성능을 향상시키기 위해 복잡한 MARK 명령어가 PDP-11에 추가되었는데, 이 명령어는 프로그래머가 원하는 것을 정확히 수행하지 못하기 때문에 거의 사용되지 않습니다. VMS FOR- TRAN 컴파일러가 잠재적인 VAX 명령어 중 매우 많은 부분(.8?)을 생성한다는 소문이 있습니

다.

VAX에서 이러한 유형의 명령어 몇 가지에 대한 모델로 FORTRAN과 BLISS가 사용되었다고 해도 놀라지 않을 것입니다. 조건부 분기를 정확하게 구현하지만 다른 많은 언어에서 흔히 볼 수 있는 0과 0이 아닌 경우의 분기에는 두 개의 명령어를 필요로 하는 조건부 분기를 정확하게 구현하지만 더 일반적인 경우의 분기에는 쓸모가 없는 브랜치 **이프 하부 비트** 세트와 **브랜치 이프 하부 비트 클리어** 명령을 생각해 보십시오. 포트란에도 이와 유사한 명령어와 주소 지정 모드가 존재합니다.

Peuto와 Shustek은 IBM 및 Airidahl 계산기의 복잡한 십진수 및 문자 명령어가 일반적으로 하이엔드 모델에서 상대적으로 성능이 저하되는 것을 관찰했습니다. 이들은 더 간단한 명령어가 성능 향상으로 이어질 수 있다고 제안했습니다[Peuto,Shustek77]. 이들은 또한 인스트럭션 쌍의 동적 발생을 측정했는데, 여기서 중요한 결과는 CISC 철학을 뒷받침할 수 있습니다. 그들의 결론은 다음과 같습니다:

"빈번하게 발생하는 옵코드 쌍을 조사한 결과, 이를 대체하기 위한 추가 지침을 만들 것을 제안할 만큼 자주 발생하는 쌍을 발견하지 못했습니다."

[7] C 컴파일러를 VAX로 포팅할 때 버그의 절반 이상과 복잡성의 약 3분의 1은 다음과 같은 원인으로 인해 발생했습니다. 복잡한 **인덱스 모드**.

RISC 아키텍처 작업

버클리 와이에서는 D.A. 패터슨과 C.H. 세퀸의 감독 하에 몇 달 동안 **RISC** 아키텍처에 대한 연구가 진행되어 왔습니다. 적절한 명령어 집합을 신중하게 선택하고 해당 아키텍처를 설계하면 매우 빠르면서도 매우 간단한 명령어 집합을 가질 수 있을 것으로 생각합니다. 이는 전체 프로그램 실행 속도에서 상당한 순이익을 가져올 수 있습니다. 이것이 바로 축소 명령어 집합 컴퓨터의 개념입니다. RISC를 구현하는 것은 거의 확실하게 CISC를 구현하는 것보다 비용이 적게 들 것입니다. 단순한 아키텍처가 고급 언어 프로그래머에게 VAX나 IBM S/38과 같은 CISC만큼 효과적이라는 것을 보여줄 수 있다면 효과적인 설계를 했다고 주장할 수 있습니다.

벨 연구소에서. C 프로그래밍 언어의 측정을 기반으로 컴퓨터를 설계하는 프로젝트는 벨 연구소 컴퓨팅 과학 연구 센터의 소수의 개인에 의해 수년 동안 연구되어 왔습니다. A.G. 프레이저가 프로토타입 16비트 컴퓨터를 설계하고 제작했습니다. 32비트 아키텍처는 S.R. 본, D.R. 디첼, S.C. 존슨에 의해 연구되었습니다. 존슨은 기계를 제안하고, 컴파일러를 작성하고, 결과를 측정하여 더 나은 기계를 제안한 다음, 이 과정을 수십 번 이상 반복하는 반복 기법을 사용했습니다. 초기 의도는 특별히 단순한 디자인을 고안하는 것이 아니었지만, 그 결과 코드 밀도가 PDP-11 및 VAX만큼 컴팩트한 RISC와 유사한 32비트 아키텍처가 탄생했습니다[Johnson79].

IBM에서. 의심할 여지 없이 RISC의 가장 좋은 예는 뉴욕주 요크타운 하이츠에 있는 IBM Research에서 개발한 801 미니컴퓨터입니다[Electronics76] [Datamation79]. 이 프로젝트는 몇 년 전부터 대규모 설계 팀이 첨단 컴파일러 기술과 함께 **RISC** 아키텍처의 사용을 모색해 왔습니다. 많은 세부 사항이 부족하지만 초기 결과는 매우 특별해 보입니다. 이들은 IBM S/370 모델 168의 약 5배 성능으로 실행되는 PL/I의 하위 집합에서 프로그램을 벤치마킹할 수 있었습니다. 더 자세한 정보를 기대하고 있습니다.

결론

특정 "고유" 명령어가 프로그램의 속도를 크게 향상시킬 수 있는 예는 의심할 여지없이 많이 있습니다. 시스템 전체에 동일한 이점이 적용되는 예는 거의 보지 못했습니다. 다양한 컴퓨팅 환경에서 명령어 집합을 신중하게 잘라내는 것이 비용 효율적인 구현으로 이어진다고 생각합니다. 컴퓨터 아키텍트는 새로운 명령어 집합을 설계할 때 다음과 같은 질문을 스스로에게 던져야 합니다. 이 명령어가 드물게 발생하는 경우, 예를 들어 수퍼바이저 호출 명령어와 같이 반드시 필요하고 합성할 수 없다는 이유로 정당화할 수 있는가? 명령어가 드물게 발생하고 합성이 가능한 경우, 부동 소수점 연산과 같

이 시간이 많이 소요되는 연산이라는 이유로 정당화될 수 있습니까? 명령어가 소수의 더 기본적인 명령어로부터 합성 가능한 경우, 해당 명령어를 생략할 경우 프로그램 크기와 속도에 미치는 전반적인 영향은 무엇인가요? 예를 들어, 사용하지 않는 제어 저장소를 활용하거나 ALU에서 이미 제공한 연산을 사용하여 명령어를 무료로 얻을 수 있습니까? "무료"로 얻을 수 있다면 디버깅, 문서화 및 향후 구현에 드는 비용은 얼마인가요? 컴파일러가 명령어를 쉽게 생성할 수 있을까요?

우리는 고급 언어 컴퓨터 시스템의 정의를 충족하면서 '복잡성' (아마도 설계 시간과 게이트로 측정)을 최소화하고 '성능'(아마도 게이트 지연으로 표현되는 평균 실행 시간을 기술에 독립적인 시간 단위로 사용)을 극대화하는 것이 가치가 있다고 가정했습니다. 특히, 우리는 VLSI 컴퓨터가 RISC 개념의 혜택을 가장 많이 받을 수 있을 것으로 생각합니다. VLSI 기술의 급속한 발전으로 인해

아키텍처의 복잡성을 개선하는 만병통치약으로 사용됩니다. 우리는 각 트랜지스터가 적어도 향후 10년 동안은 귀중할 것으로 보고 있습니다. 아키텍처 복잡성을 향한 추세가 향상된 컴퓨터를 향한 한 가지 경로일 수 있지만, 이 백서에서는 또 다른 경로인 명령어 집합 감소 컴퓨터를 제안합니다.

감사

이 백서에 대한 신속하고 건설적인 의견을 보내주신 A. V. Aho, D. Bhandarkar, R. Campbell, G. Corcoran, G. Chesson, R. Cmelik, A.G. Fraser에게 감사의 말씀을 전합니다,
S.L. 그레이엄, S.C. 존슨, P. 케슬러, T. 런던, J. 오스터하우트, D. 포플러스키, M. 파월, J. 라이저, L. 로위, B. 로랜드, J. 스웬센, A.S. 타넨바움, C. 세퀸, Y. 타미르, G. 테일러, 및
J. Wakerly. RISC 프로젝트에 참여한 버클리의 학생들은 우수한 연구 성과를 인정받았습니다. 버클리의 RISC 연구는 부분적으로 국방부 고등연구계획국(DoD), ARPA 주문 번호 3803의 후원을 받았으며 계약 번호 N00039-78-G-0013-0004에 따라 해군 전자 시스템 사령부에서 모니터링했습니다.

참고 자료

[Alexander75]

[Shustek78]

[Utley78]

[코케80] [크래곤80]

[데이터메이션79] [

디체1,패터슨80]

[전자76] [포스터7l] [

허슨70] [존슨79] [패

터슨,세퀸80]

[Peuto,Shustek77]

W.C. Alexander와 D.B. Wortman, "XPL 프로그램의 정적 및 동적 특성", *컴퓨터,* 41-46쪽, 1975년 11월, 8권, 11호.

J. Cocke, *개인 통신,* 1980 년 2 월.

1980년 5월, 고급 언어 컴퓨터 아키텍처에 관한 국제 워크숍에서 '고급 언어 컴퓨터에 대한 사례'라는 논문을 발표하는 H.A. Cragon의 강연.

데이터메이션, "IBM 미니의 급진적 출발", 1979년 10월, 53-55쪽.

"고수준 언어 컴퓨터 *아키텍처에* 대한 회고", *제7회* ***컴퓨터*** *아키텍처 국제* ***심포지엄,*** 1980년 5월 6-8일, 프랑스 라 바울.

Electronics Magazine, "컴퓨터 아키텍처를 변경하는 것이 처리량을 높이는 방법이라고 IBM 연구원들은 제안합니다.", 1976년 12월 23일, 30-31페이지.

C.C. Foster, R.H. Gonter, E.M. Riseman, 'Measures of Op-Code Utilization," ***IEEE Transactions on Computers,*** May, 1971, 582-584쪽.

S.S. 허슨, ***마이크로프로그래밍: 원리와 관행,*** Prentice-Hall, Engelwood, N.J., 109-112쪽, 1970.

S.C. Johnson, "32비트 프로세서 설계", 컴퓨터 과학 기술 보고서 80호, Bell Labs, Murray Hill, 뉴저지, 1979년 4월 2일.

D.A. Patterson 및 C.H. Séquin, '설계 고려 사항: 미래의 단일 칩 컴퓨터를 위한 설계', ***IEEE*** *Journal of Solid-State* ***Circuits, IEEE Tran-*** *sactions* on *Computers,* 공동 특별호 마이크로프로세서 및 마이크로컴 퍼터, C-29, 2권, 108-116페이지, 1980년 2월.

B.L. Peuto와 L.J. Shustek, "CPU 성능의 명령 타이밍 모델," *Conference* ***Proc.*** *, 제4차 컴퓨터 아키텍처 연례 심포지엄,* 1977년 3월.

L.J. Shustek, "컴퓨터 명령어 세트의 분석 및 성능," 스탠포드 선형 가속기 센터 보고서 205, 스탠포드 대학교, 1978년 5월, 56페이지.

B.G. Utley *외,* "IBM 시스템/38 기술 개발," IBM GS80- 0237, 1978.

ERRATA

29페이지 하단의 각주 "Peduced 7n 명령어 집합 컴퓨터의 경우"에 오류가 있습니다. 하비 크래건은 컴퓨터를 더 복잡하게 만들어 의미적 격차를 줄이는 것에 대해 이야기하고 있었습니다. 가장 적절한 슬라이드는 바로 아래에 인용되어 있습니다:

> "... 벡터 연산을 수행한 결과 이 두 시스템(CoC 7SP0과 TI ASC)의 성능은 거의 동일했습니다. 메모리 대역폭은 거의 동일했습니다. 하드웨어 D0 루프의 버퍼링은 일반 명령 스트림의 버퍼링과 동일한 메모리 대역폭 감소를 달성했습니다.
> >°00. 에 대한 적절한 매크로가 작성된 후
> 7 AO와 컴파일러에 통합된 호출 프로시저를 통해 벡터 기능에 대한 동등한 액세스가 포트란에서 제공되었습니다. 벡터 하드웨어에 의해 도입된 복잡성 때문에 ASC의 스칼라 성능은 7G0P보다 낮습니다. 마지막이자 가장 중요한 논거는 7G0M에 필요한 것보다 ASC에 더 많은 하드웨어가 필요하다는 것입니다. 다소 지루한 DO 루프 연산을 구축하기 위해 추가한 하드웨어는 예상했던 만큼의 성과를 거두지 못했습니다.
>
> ASC와 다른 경험들을 통해 저는 의미적 격차만 줄인다면 약속한 모든 장밋빛 약속에 의문을 갖 게 되 었 습 니 다 . 약속한 혜택은 실현되지 않으며, 일반성은 가장 중요한 것입니다."

이 두 가지의 상대적인 속도는 주어진 애플리케이션에서 스칼라 연산과 벡터 연산의 조합에 따라 달라집니다.

또한 실수로 Harvey의 중간 이니셜을 변경하여 Harvey C. Cragon으로 변경했습니다.

데이브 패터슨
데이브 디첼