\*불이 컴퓨터에 끼치는 영향

-Mathematical logic 분야 열다 (빈틈없는 논리의 전개)

-> AI: 지식(명제)의 저장 및 논리적 추론할 수 있는 기계를 개발. 컴퓨터가 지능이 있는 기계로 개발에 대한 궁금증에서 메모리(memory)에 사람의 지식과 논리적인 추론 능력을 담으면 지능이 기반이 된 기계를 만들려 했다. 이것이 knowledge-based approach의 핵심이며 명제 형태의 지식을 저장하고 논리적 추론을 하는데 이론적인 기반을 제공한 것이 mathematical logic이다.

-불 대수학(Boolean Algebra)

사칙연산을 사용하지 않고 AND, OR, NOT을 사용하여 0,1을 참과 거짓으로 사용하여 이진 값들로 표현하였다. 사칙 연산(Arithmetic) 과 컴퓨터에서 핵심 역할을 하는 CPU가 계산과 논리적인 처리할 수 있는 ALU를 할 수 있다.

* 자동장치 설계(Automata Design): 댐의 수문에 대한 간단한 자동장치를 설계를 할 때에도 해당 작용을 명제로 다루며, 불(Boole)의 명제들을 사용하면 논리적인 다이어그램(logic diagram)을 쉽게 구현할 수 있다.
* 컴퓨터 과학의 개념적인 기초 설계

Boolean algebra는 digital logic design으로 연결되어 logic gate를 이용해 컴퓨터를 만들 수 있게 해주었으며 컴퓨터 과학의 개념적 기초를 만들어주는데 영향을 끼쳤다.

\* Combinational logic design and Sequential Logic Design

-Combinational logic design (입력이 정해지면 결과가 명확하게 결정되는 논리)

결과 값이 순수하게 입력 값만으로 결정되는 논리 디자인으로 입력 값에 따라 결과 값이 항상 똑같이 결정된다.

-Sequential Logic Design

결과 값이 입력 과 state or register로 결정되는 논리 디자인이다(input만으로 결정되는 조합 논리보다 복잡하다)

자체적으로 메모리에 정보를 담고 있는 그것을 state라고 한다. 결과 값은 입력 값과 state 값에 의해서도 결정되며, next state값도 input과 현 state값에 따라서 결정된다.

\* Abstraction

추상화라는 개념은 근본적으로 사용법(Interface)와 구현(Implementation) 개념이 담겨있다. 컴퓨터 과학에서 두가지 중요한 인터페이스를 보자면 컴파일러(Compilers) 와 프로그래밍 언어(ex>c, c++, java)가 있는데 여기서 컴파일러는 구현에 해당하고 c와 같은 프로그래밍 언어는 고급 언어 사용법 즉, interface에 해당한다

primitive-composition-abstraction

c programming

primitive: statement (variable, assignment, arithmetic and assignment, conditional, Loop, compound), function call statements -> C statement에서는 산술연산과 조건을 사용할 수 있으며 영어와 같은 제어 구조도 사용 가능하다. 필요한 statement를 정교하게 완성

composition: c의 statement를 사용하여 function을 만든다. 우아한 function decomposition

function abstraction: 프로그램의 크기와 복잡성을 다루는데 중요하다. function을 만들어 놓으면 이것은 곧 primitive가 되고 나중에 c의 statement가 되어진다. 성능 고려한 논리적인 api

\*\*Library: 함수들의 집합, API: C에서는 헤더파일

OOP

OOP는 한가지 메커니즘이 추가되는데 object를 추가되며 다른 function보다 커다란 데이터와 프로세싱의 추상으로 결합되어진다.

처음에 언급한 function 보다 훨씬 object 단위로 abstraction이 일어난다. >> object libraries

c 프로그래밍 실행 되는거는 func, statement가 하나하나 연속적으로 실행된다는 개념이고

OOP는 많이 필요한 object를 만들고 object간에 message를 주고받는 분산 시스템 개념을 갖고 있다.

\* Fetch – Decode – Execute

컴퓨터의 메모리가 외부에서 내부로 옮겨지면서 컴퓨터의 동작원리는 fetch-decode – execute가 되었다.

\* Platform Dependency

프로그램 실행환경 : 프로세서가 제공하는 instruction set , OS가 API interface를 합한거.

single-chip

2차 대전 후에 많은 반도체 회사 중 하나인 인텔은 칩 주문 설계 및 생산하여 컴퓨터는 미국 동부가 주도하고 있었다. 한 고객이 계산이 특화된 칩을 인텔에게 요구하였지만 인텔에서는 범용 프로세서를 싱글 칩으로 만들어 4004 마이크로프로세서 칩을 만들었다. 범용 프로세서를 칩으로 만들면 계산할 수 있는 전용 칩을 만들 필요 없이 프로그램으로 만들면 되기 때문에 더욱 효율적이었다. 이것은 후에 컴퓨터 회사가 되었다. 1970년의 메인 프레임의 프로세서들은 32비트로 시작하였지만 IC 기술이 발전이 되어있지 않았기 때문에 32-bit 컴퓨터를 구성하기 위해서 많은 칩들을 필요했기 때문에 PCB(printed circuit boards)가 대량으로 필요했다. 그런 시대에서는 인텔에서는 4비트 프로세서를 만들었다. 후에 8비트 프로세서가 상용하게 되면서 마이크로 컴퓨터가 만들어지기 시작하여 소프트웨어와 마이크로 컴퓨터가 대중화되었다. 싱글 칩 프로세서들은 강력한 프로세서로 트랜지스터의 규모와 향상된 설계가 더해졌고 컴퓨터 회사들은 프로세서 공급업체로부터 프로세서를 구입하여 1980년도에 컴퓨터 회사에 거대한 일자리에 대한 변화를 가져다 와줬다. 이후 시스템과 소프트웨어, 서비스에 집중하게 되었고, 작은 마이크로프로세서는 여전히 시스템이 탑재되어 적은 비용으로 남아있다.

ISA compatibility (and golden shackle)

인텔의 마이크로프로세서 칩이 점점 발전할 때마다 이전의 IS를 수용하여 다음 IS를 만든다. 그 이유는 이전의 IS를 사용하는 사용자가 새로운 IS 프로그램을 쓰지 못하면 해당 회사의 브랜드에 대한 충성도가 낮아지는 것을 우려해 인텔은 IS를 발전해 나갈 때마다 이전 IS를 수용하여, 이전 IS도 새로운 IS 프로그램에 대해 문제가 없이 업데이트를 해준다.

\*컴퓨터 성능 발전 요인

반도체 & 설계기술 요인

\* Trends in Technology -> exponential

프로세서 기술은 exponential 성능 증가를 보여주는데, 하나는 반도체의 트랜지스터 밀집도에 따라서 exponential 성능 증가(35 %/year)를 하였고, die 크기로 exponential 성능 증가(10-20%/year) 성능 증가 설계를 위해서 die 크기가 증가하게 된다. 매년 40-55 %의 exponential 성능 증가(speed)가 일어났다. 메모리에서는 dram, 플래시 메모리, 하드 디스크 메모리가 수용 능력이 exponential 용량 증가가 일어났다.

\* 64비트 컴퓨터가 32비트 컴퓨터보다 더 좋은 점은 무엇인가? Address의 크기가 컴퓨터에서 왜 중요한가?

x-bit computer는 ALU에 input에 32 bit 정보가 들어와서 처리하는 것을 말한다. 64-bit computer가 32-bit computer 보다 좋은 이유는 더 큰 수를 계산할 수 있다. 그리고 동일한 작업에 대해서 더 빠르게 작업을 할 수 있다. 그 다음으로 중요한 것은 address line이다. address의 bit는 작업할 수 있는 프로그램의 최대 크기가 얼마인가에 대해 밀접한 관계가 있다.

\*alignment

Alignment를 요구하는 architecture에서는 half word를 저장할 때 짝수 번지에서만 시작하는 것을 허용합니다. 홀수 번지에서 시작하여 저장했을 경우 데이터를 나눠서 저장하게 되고 접근할 때 2번 해야 하기 때문입니다. 한 word를 저장할 때도 마찬가지로 4의 배수인 번지에서만 허용합니다. 그래서 alignment는 메모리 access speed와 관련이 있습니다.

\* Interrupts + 동작 방식

programmed i/o vs interrupt

프로그램에서 입출력 장치들을 간헐적으로 사용을 한다. 그렇다면 프로세서를 수행하는 와중에 입출력 장치의 작업을 수행하고자 할 때에는 어떻게 해야 하는가?

CPU는 프로세서와 입출력장치에서 신호가 오면 해당 작업을 수행시켜 줘야 한다. 그래서 CPU에서는 입출력장치와 단일적으로 연결이 되는 INT line이 존재를 한다. INT line에서 신호가 0으로 시작하여 입출력 장치에서 어느 작업이 필요할 때에는 INT line 신호를 1로 바꾸어 CPU에 신호를 보낸다. INT line을 만들어 줌으로써 CPU는 별개로 입출력 장치 신호에 신경 쓸 필요가 없다. 주기적인 입출력 장치를 점검할 필요가 없어진 것이다. 프로세서는 추가적인 하드웨어를 가지고 있어서 INT가 걸리면 정해진 위치로 뛰게(JUMP) 되어있다. 그 자리에 입출력 장치 처리 routine을 넣어 두면 된다. 입출력 신호에 대한 루틴을 ISR(interrupt service routine) 이라 부른다.

프로세서는 interrupt에 대해 입출력을 정기적으로 점검할 필요가 없다. 해당 pc위치가 작업한다고 가정할 때, 입출력 신호가 올 경우 jump를 하는데 그 전에 해당 PC값을 저장하고 사용한 Register 값도 저장한다. 그 다음에 JUMP를 하여 ISR위치로 이동하여 입출력 작업을 마무리시키면 IRET(return from INT) 명령어를 써서 저장하였던 레지스터와 PC값을 복원한다

Multiple INTs and INT Priority

지금까지 interrupt에 하나에 대해 언급을 하였지만 지금부터는 여러 개의 interrupt가 들어올 경우를 생각해본다. 시스템을 설계할 때 정하게 입출력장치의 우선순위를 할당해 줘야한다.

\*Atomic Operations 0-5

enable INT, disable INT 명령을 쓸 경우 이 것은 작은 임베디드 시스템에서 사용된다. 작은 임베디드 시스템에서는 프로그램이 한 종류로 고정되어 있고 프로그램이 하드웨어를 컨트롤을 한다. 그런 경우 disable INT 명령을 쓸 경우 아무리 critical sections 또는 트랜잭션이 작업이 들어와도 해당 작업이 끝나야 INT 작업이 실행시킬 수 있다. 자원을 공유하는 소수의 프로그램에서 발생하는 문제에서 해결하기 위해서 atomic operations을 작동시킨다. 범용컴퓨터에서 실행시키면 자원을 공유하지 않은 관계없는 프로세서에서도 적용되기 때문에 프로세서 스케줄링을 할 수 없어진다. 그래서 범용 컴퓨터에서는 atomic operations을 사용하지 않는다.

복잡한 소프트웨어는 여러개의 모듈을 만들고 서로 소프트웨어 모듈끼리 request를 한다.

하지만 서로 소프트웨어끼리 공유를 할 때 problem이 생기기 마련이다.

어떤 소프트웨어 모듈이 아토믹 서비스를 보장을 한다면 recoverability가 보장이 된다.

\* 제품 개발 과정(iterative)

1.마케팅, 요구사항 분석, 계획

2. design 그리고 implementation

(isa 설계 ic -> high-level cpi -> lower-level circuits cct -> overall performance) (iterative)

3. 테스트

4. 연속적인 보완

\*어떤 것을 측정을 해야 좋은 컴퓨터인가?

수행 측정

what do we measure for Computer Performance?

\*프로세서의 개선을 통한 response time과 throughput의 개선

1. Response time: 작업을 끝 마치는데 얼마나 오래 걸리는가?
2. Throughput: 하나의 단위 유닛을 마무리하는데 걸리는 전체 시간

우리는 응답 시간에 보통 초점을 둔다. 왜냐하면 Throughput 자체의 성능을 높이려면 대규모 설계가 필요하다. 하지만 Response time을 줄이면 자동적으로 Throughput도 동시에 성능이 좋아지기 때문에 Throughput보다는 성능 개선이 편한 Response time에 초점을 둔다.

How do we measure for Computer Performance of Evaluation methods?

컴퓨터 수행을 평가하는 방법으로 실제 응용 프로그램을 대표할 수 있는 프로그램이 선정 되어있다. 그것을 Benchmarking이라 불리는데 Benchmarking으로 Response time을 측정한다.

How can we improve for Computer Performance?

IC(instruction count)

* machine instruction이 실행되는 수로 동적인 수라고 말한다. 동적인 수(Dynamic(run time) count)란? 실제 실행되는 instruction의 수로 개발자가 작성한 코드에는 많은 machine instruction이 있다. instruction 마다 실행이 되는 것과 실행이 안되는 것이 존재한다. 또는 Loop 라는 instruction이 존재하는데 만약 n이 10000이라면 이 Loop라는 IC는 10000이 된다. 만약 n이 input data 라면 해당 프로그램은 input data마다 IC가 변한다. 여기에서 Input을 고정하거나 여러 input에 대한 평균을 취하여 IC를 check을 한다.
* 최상위 개념 IC는 ISA(instruction set architecture)(I/F) 설계에서 결정되며, ISA가 확정이 된다면 컴파일(Compiler) 설계가 가능하다.

CPU 시뮬레이터로 Benchmark을 돌리면 IC를 얻고, 이에 따라 implementation(CPI & cct)가 영향을 받는다.

CPI(cycle per instruction)

* 차상위 개념, high-level implementation – 한 instruction이 실행되는데 필요한 clock cycles의 수로 instruction 마다 다르다.
* 최종적인 CPI는 빈도에 따른 weighted average로 계산한다. – 실제 실행되는 Dynamic count를 고려한다.
* 설계된 ISA 구현시 CPI가 결정된다.

\*Pipelining: 3-Stage Pipeline

첫 번째 그림에서 CPI는 1이고, cct는 9ns 로 가정하자. 해당 CPI를 3등분을 하면 자동적으로 cct도 3등분이 되어 진다. 그렇다면 해당 CPI와 cct를 가지고 3-stage로 Pipeline을 설계를 한다면 한 cycle에 instruction의 수는 1이 되고 cct는 3ns 이기 때문에 IC는 3 ns 가 된다.

cct(clock cycle time)

하위 개념으로 clock 한 주기의 길이를 의미한다. ISA 및 high level implementation이 결정되면, 한 클럭 내에서 할 일 들일 결정되는데 cct를 줄일수록 성능이 좋아진다.

회로 설계자는 cct를 줄이려 노력을 하는데, clock 내에 할 일들 중에서 가장 오래 걸리는 작업을 더욱 빠르게 만들려고 노력을 한다. 1clock cycle은 이미 결정되었지만 그 안에 signal을 줄이려고 노력을 한다.

12

GPR Architectures

Three types

•Register-register architecture

ALU는 레지스터의 내용을 기반하여 연산을 한다. ALU의 연산의 INPUT 데이터는 레지스터에서 온다.

ALU의 필요한 데이터들이 레지스터에 존재하지 않으면 메모리에서 읽어서(load) 연산을 한다.

load나 store를 통해서만 메모리 접근

•Register-memory architecture

ALU instruction에 메모리를 허용한다.

•Memory-memory architecture

ALU 연산에서 바로 메모리에서 데이터를 읽어서 , 두 개의 INPUT이 메모리에서 읽어서 결과값을 ALU 자체에서 메모리에 저장할 수 있다.

Rationale behind CISC

응용프로그램의 크기를 줄이는데 중요시하였다. 왜냐하면 i/o 시간이 줄어드기 때문에 성능면에서 좋아지기 때문에.

응용프로그램 파일은 상당히 커졌다 .하지만 메인 메모리는 그것 보다 작았기 때문에 응용프로그램의 일부 데이터를 사용할 수 있었다.

만약 응용프로그램을 또 다른 부분을 실행시키기 위해서는 다시 메인메모리에 다시 load해야하기 때문에 상당히 많은 시간이 걸리게 되었다.

RISC보다 CISC에서 많은 INSTRUCTION을 사용함으로써 메모리를 적게 쓰게되어서 당시에는 효과적이였다.

amdahl`s law 에서는 멀티플리케이션을 2배, 4배 성능를 빠르게하면 전체적인 성능 향상이 보이지만 10배 , 무한대 빠르게하면 속도를 빠르지만 ,멀리플리케이션이 굳이 모든 연산에서 자주쓰이는 것이 아니기 때문에 , amdahl`s law의 핵심은 자주 쓰이는 연산을 빠르게하는게 핵심이다.

DIE 사이즈를 두배를 높이면 성능은 두배가 아니라 1.4배 즉 40% 증가하고, POWER은 두배가 높아진다..

POWER에서 문제가 생김.

DIE 사이즈를 증가시키는것은 무리가 있다. 그래서 프로세서를 단순한 프로세서를 써서 해당 다이 하나에 단순한 프로세서를 4개를 넣음으로써 , 즉 단순한 프로세서안에 또다른 프로세서를 넣어서 MULTICORE단