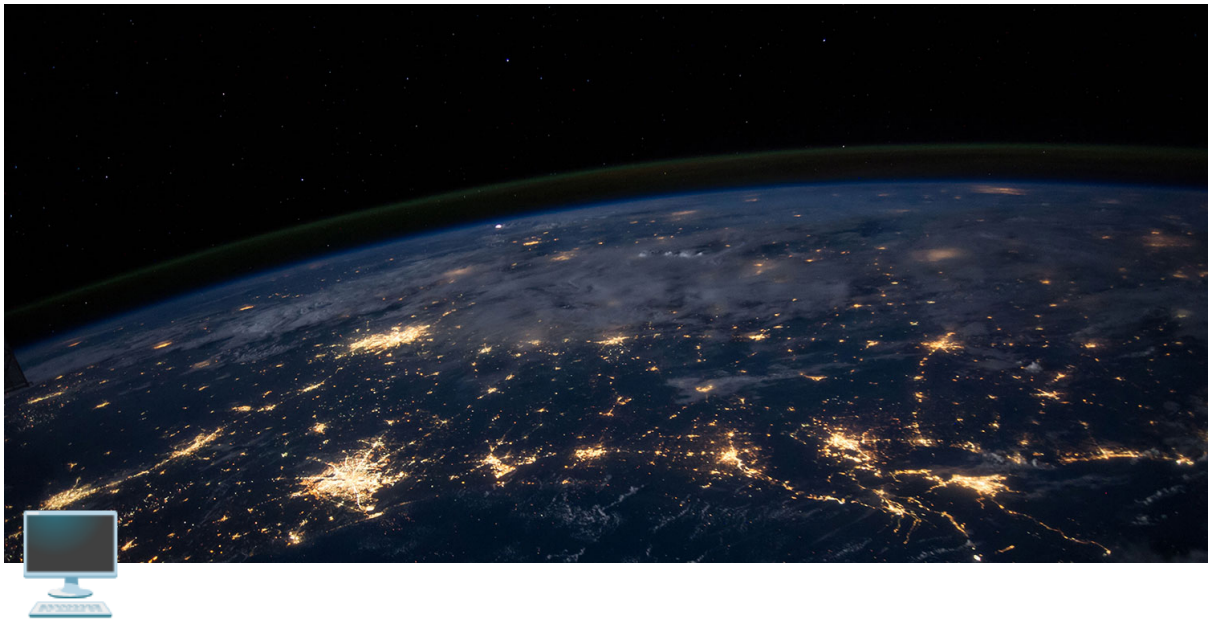
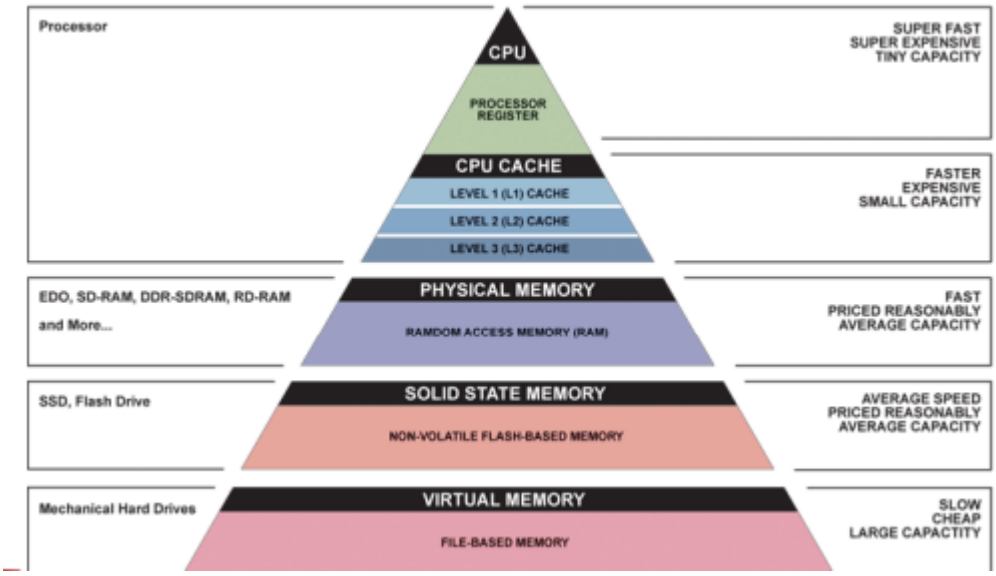


Chapter_01.Introduction

What you will run

1. 어떻게 고급 언어 프로그램이 하드웨어상에서 돌아가는 걸까?
2. 소프트웨어와 하드웨어 사이의 관계는 뭘까?
3. 수행 능력을 결정짓는 것은 무엇일까?
4. 어떻게 수행 능력을 향상시킬 수 있을까?
5. 최근에 순차 처리에서 병렬 처리로 넘어가고 있는 이유와 결과가 무엇일까?

Important Concepts In Computer Architecture (컴퓨터 아키텍처에서의 중요한 컨셉)



1. Moore's Law를 따라 디자인을 하도록 하자.

- 무어의 법칙은 반도체 성능이 2년마다 2배로 증가한다는 법칙이다.

    1. 단순하게 디자인하기 위해 추상화를 사용해라.

- 높은 단계를 구현하기 위해 무조건 낮은 단계가 어떻게 작동하는지 알아야 하는 것은 아니다.
  예를 들어, CPU를 설계하는데 있어서 논리 게이트가 어떻게 작동하는지 모든 것을 고려한다면,
  머리가 터질 것이다.

- 레이어로 표현해라.
  3.주로 사용하는 것을 빠르게 설계하라

- 관계된 아이디어 : Amdahl's Law
  예를 들어, CPU를 자주 사용하는데 성능 향상을 위해 하드 디스크를 더 사는 것은 멍청한 짓이다.

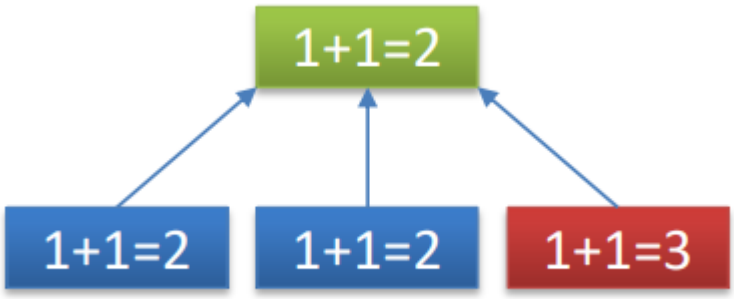
- 이와 관련된 법칙으로는 Amdahl's Law 가 있다. 이 법칙에 의하면, 병렬 처리에서 프로세스수를
  추가한다고 처리 속도가 무조건 비례하는 것은 아니다.

1. 컴퓨터 성능 향상을 위한 기술

- 컴퓨터 성능 향상을 위한 다양한 기술들이 존재한다.

1. 병렬화
2. 파이프 라이닝 -> 가장 중요
3. 예측

4. 메모리 계층
5. 중복을 통한 신뢰성



- 같은 일을 여러 개의 컴퓨터에서 한 후 중복이 가장 많은 결과를 답이라고한다. 이를 통해
  오류가 발생해서 답이 틀릴 확률을 낮출 수 있다.

Below Your Program

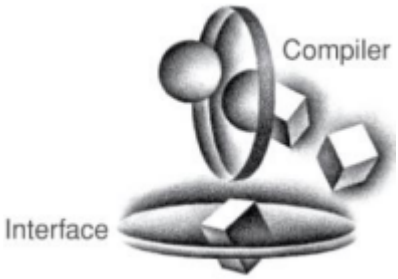
- 프로그램 밑단에선 어떤 일들이 수행되고 있을까.
  Word processor, database, ... etc,-.
- 수 만 줄의 코드가 있지만 하드웨어는 오로지 low - level에서만 작동한다.
- 단순한 인스트럭션스를 사용하자.(ADD, LOAD, STORE, JUMP, … etc)
- gap사이에 여러개의 층이 존재한다.
- 프로그램 밑단에서는 내가 사용하고 있는 워드 프로세서나 데이터베이스 등의 어플리케이션은 수 만줄의 코드로 구성되어있다.
- 이 수 만 줄의 코드들은 고급 언어로 이루어져 있으며 컴퓨터가 이 고급 언어를 바로 이해하고 실행 할 수는 없다. 컴퓨터는 0 과 1로만 이루어진 기계어만 실행 할 수 있다.
- 이를 위해서는 고급 언어를 단순한 컴퓨터 명령어로 번역해야한다. 이 역할은 어플리케이션 소프트웨어와 하드웨어 사이의 시스템 소프트웨어(운영 체제, 컴파일러) 등이 담당한다.
- 단, 어플리케이션은 하드웨어에 직접 명령을 내릴 수 없고 시스템 콜을 통해서 간접적으로 명령을 내릴 수 있다.

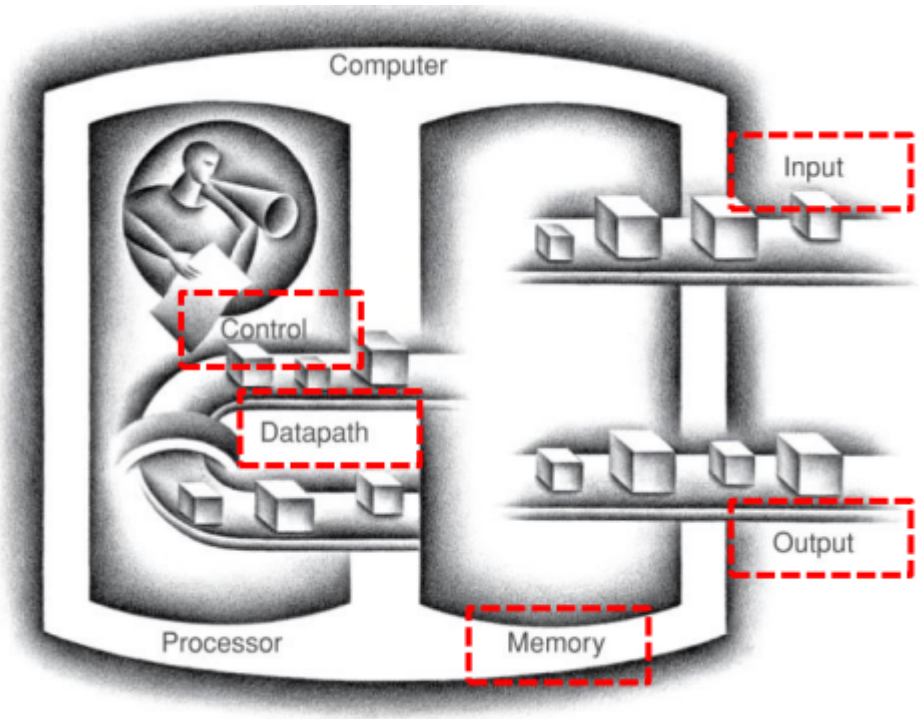
```
High-level Language -> Instructions
High-level language program(in C)
swap (int v[], int k)
{int temp;
```

```
temp = v[k]:
v[k] = v[k+1]:
v[k+1] = tmep;
}
```

- -> 컴파일러 -> 어셈블리 언어 프로그램 (for MIPS) -> Low level programming language, -> Assembler-> Binary machine language program(for MIPS)

Components of a Computer





- 컴퓨터의 5가지 구성요소 : Control, Datapath, Memory, Output, Input
- 프로세서 : Datapath+Control
- Input/Output 은 다음의 것들을 포함한다.
    1. User-interface devices
        a. Display, keyboard, mouse
    2. Storage devices
        a. Hard disk, CD/DVD, flash
    3. Network adapters
        a. 다른 컴퓨터들과 소통하기 위함이다.

1. & 2. Input/Output device
    a. 모든 컴퓨터는 입력과 출력이 가능해야 한다. 입력 장치에는 키보드나 마우스가 대표적이다. 출력 장치에는 모니터나 스피커가 대표적이다. 하드 디스크나 CD/DVD, USB 같은 저장 장치나 네트워크 어댑터도 입출력 장치라고 할 수 있다.
2. Memory
    a. 메모리는 프로그램이나 프로그램을 실행하는데 필요한 데이터를 저장하는 장치이다.
3. Processor
    a. CPU또한 프로세서이다. 프로세서는 컴퓨터의 가장 중요한 부분으로, Datapath와 Control로 이루어져 있다. 프로세서는 메모리부터 명령어와 데이터를 받는다.
    b. Datapath: 실제로 데이터를 처리하거나 수학적인 연산을 하는 장치다.
    c. Control: 데이터를 처리/연산하기 위해, 명령어를 해석하거나 흐름을 제어하는 장치이다.

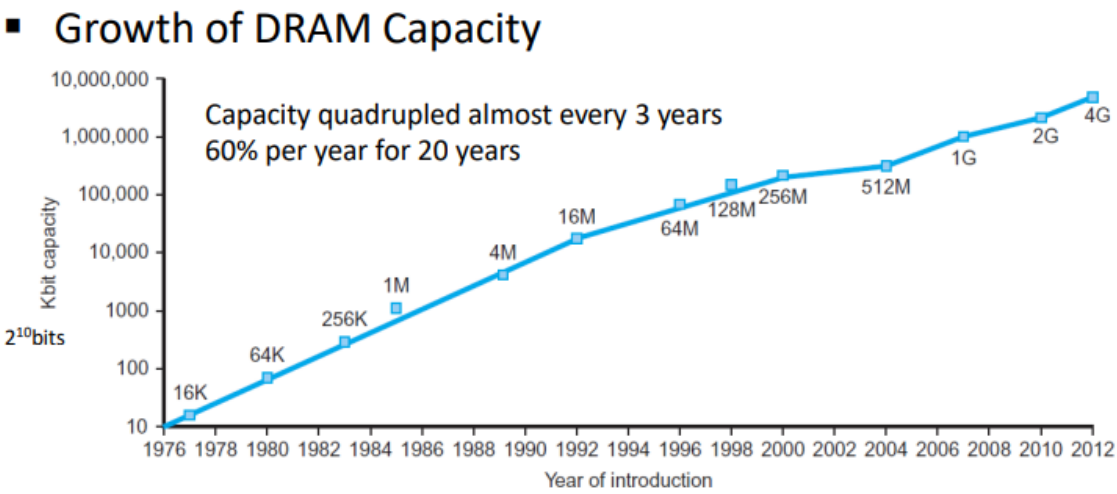
- 주의 할 점 : Interface는 컴퓨터의 5개의 기본적인 컴포넌트가 아니다. Input 과 Output을 interface와 착각하지 말도록 하자.

Technology Trends

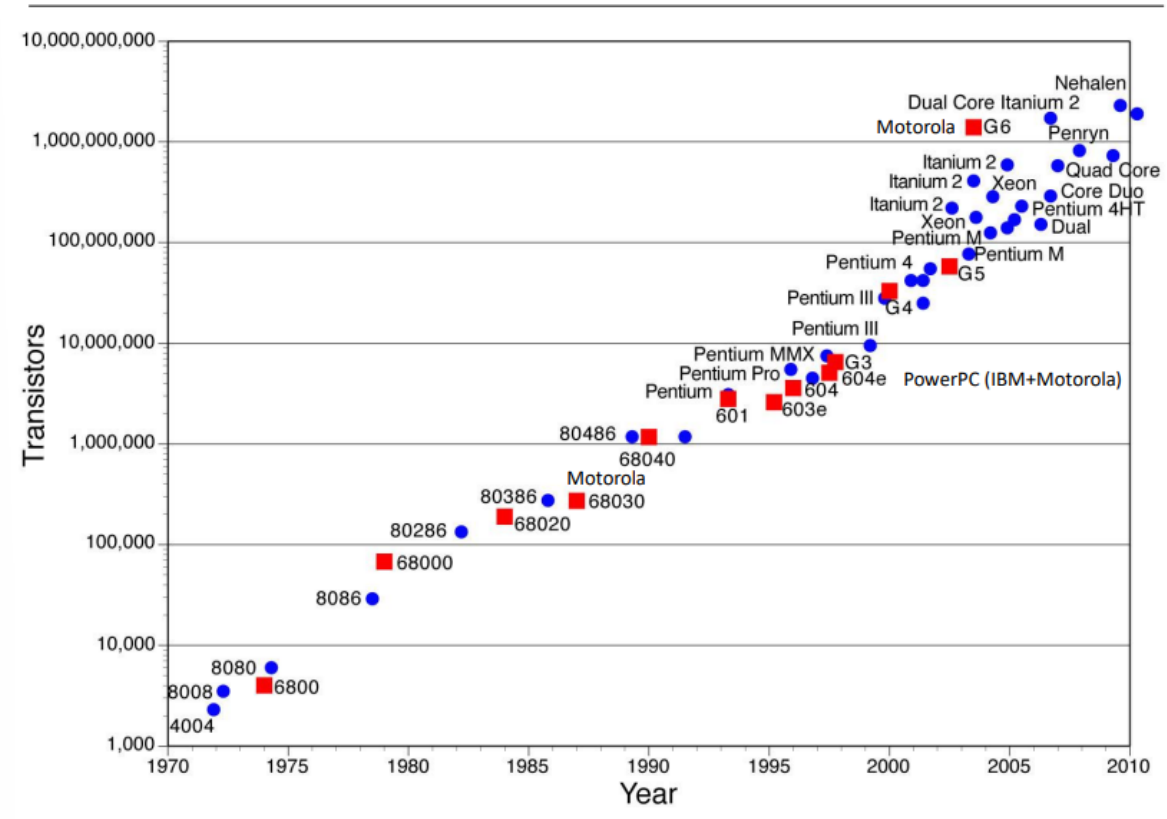
- 과거부터 오늘날까지 기술 트랜드에 대해 알아보자.

Year	Technology used in computers	Relative performance/unit cost
1951	Vacuum tube	1
1965	Transistor	35
1975	Integrated circuit	900
1995	Very large-scale integrated circuit	2,400,000
2013	Ultra large-scale integrated circuit	250,000,000,000

- 현재까지 프로세서와 메모리의 속도가 엄청난 속도로 발전해왔다. 1951년에 사용되던 진공관의 속도를 1 이라고 하면, 현재 프로세서의 속도는 약 250,000,000,000 배 이다.
- DRAM의 용량은 20년 동안 매년 60%씩 증가해왔다.



Moore’s Law



- 무어의 법칙 : 프로세서 성능이 2년마다 약 2배로 증가한다는 법칙이다. 1965년 인텔의 공동 설립자인 고든 무어가 제시했다.
- 그러나 현재 멀티 코어로 넘어오면서 법칙이 깨지기 시작했다.

컴퓨터 성능 (Perfomance)

- 컴퓨터는 매우 복잡한 장치들로 구성되어 있기 때문에, 컴퓨터의 성능을 측정하는 것은 매우 어려운 일이다.
- 컴퓨터 성능에서 주요한 측정 값 두 가지가 있다.
  1. 실행 시간 또는 응답 시간
     a. 시작부터 끝까지 걸리는 시간
  2. 처리율
     a. 주어진 시간 동안 처리된 일의 총량

  참고로 실행시간과 Through시간

성능의 정의

💡 지금부터 간단한 설명을 위해 처리율을 제외한 응답 시간에 집중해 봅시다.

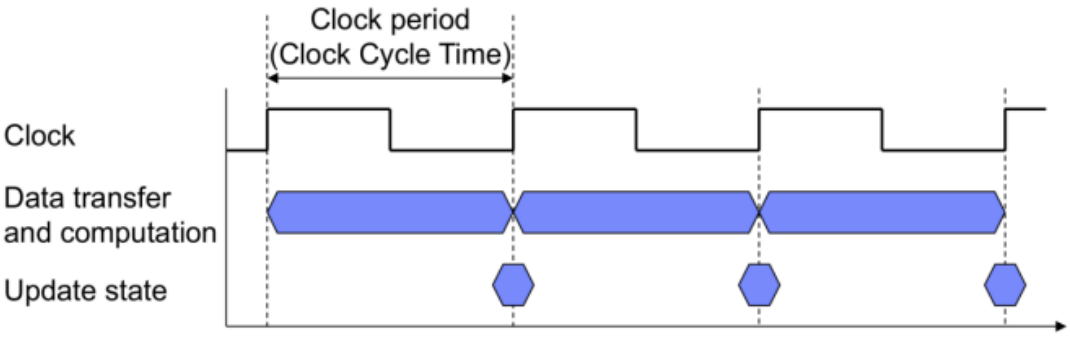
💡 $P = 1/E$, P: Performance, E: Execution time

- $P_x > P_y$식
- $1/E_x > 1/E_y$
- $E_x < E_y$
- **x는 y보다 n만큼 빠르다.** → $P_x/P_y = E_y/E_x = n$

시간측정

- 일을 수행하는데 걸리는 시간을 측정하는 것
- CPU time vs I/O time
- Clock cycle(ticks, clock ticks, cycles …)
  - 일정한 속도의 clock로 제어되는 디지털 하드웨어의 작동이다.
  - Clock Cycle

    - CPU 는 한 clock 동안 메모리에서 명령어를 가져와 해석하고, 처리한다. 이 clock의 시간 간격을 clock cycle 라고 한다.



- Clock frequency(rate): cycles per second : 단위 시간 동안의 clock cycle의 수
  - e.g., 4.0GHz = 4000MHz = 4.0x10.9Hz = 1/(250x19-12)
- Clock period (Clock Cycle Time) : clock cycle의 주기

- duration of a clock cycle = 1/clock frequency
- e.g., 250ps = 0.25ns = 250x10^-12s

CPU 성능 방정식

$$ClockCycleTime = 1/ClockRate$$

$$CPUtime = CPUclockcycles * ClockCycleTime$$

$$CPUtime = CPUclockcycles/ClockRate$$

💡 IC (instruction count) : Number of instructions executed by the program

💡 CPI(Clock Cycles Per Instruction) : Average clock cycles per insstructions for a program(fragment)

- CPI = Clock Cycles / Instruction Count
- CPU clock cycles = IC x CPI
- CPU time = IC x CPI x Clock Cycle Time = ICxCPI/Clock Rate

Using CPI(Clock cycles Per Instruction)

ex) Computer A : clock cycle time of 250 ps and CPI of 2.0

Computer B : clock cycle time of 500 ps and CPI of 1.2

Which computer is faster>? N = IC

—> CPU time A = 500N ps

—> CPU time B = 600N ps

So) CPU performanceA/CPU performanceB = Exex TimeB / Exec TimeA = 600N/500N = 1.2

Computer A is 1.2 times faster than Computer B

Comparing Code Segments

CPI Information

	CPI for each instruction class		
	A	B	C
CPI	1	2	3

Code segment Information

Code Sequence	Instruction counts for each instruction class		
	A	B	C
1	200	100	200
2	400	100	100

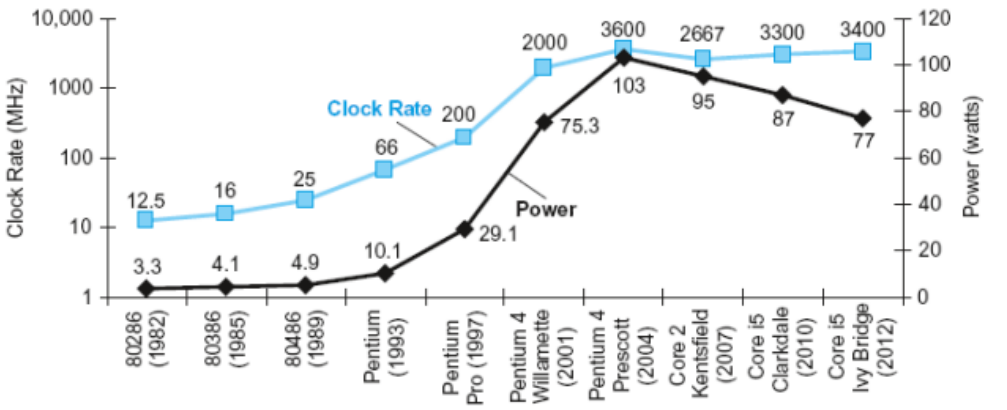
- 1 과 2 중 어떤 code sequence 가 더 많은 명령어를 실행할까?
  - segment1: 200+100+200=500instructions
  - segment2: 400+100+100 = 600instructions
  - 더 많은 명령어를 실행하는 segment2가 실행 시간이 더 길어 보인다.
- 1 과 2 중 어떤 code sequence 가 더 빠를까?
  - segment1 : 200x1+100x2+200x3=1000cycles
  - segment2: 400x1+100x2+100x3= 900cycles
  - 실행시간이 길어보이던 segment2가 사실 더 빠르게 명령을 수행한다.
- 각 segment의 CPI를 구해보자
  - segment1 : 1000/500 = 2.0CPI
  - segment2: 900/600 = 1.5CPI

Performance Summary

- CPU time = IC x CPI x Clock Cycle Time
- CPU time = (Instructions/Program) * (Clock Cycles/Instructions) * (Seconds/Clock Cycle) = Seconds/Program
- Performance difference can arise from(성능이 바뀔 수 있는 다양한 이유 4가지)
  - Algorithm(IC, CPI에 영향을 끼친다.)

- Programing language(IC, CPI에 영향을 끼친다.)
- Copiler(IC , CPI에 영향을 끼친다.)
- ISA(IC, Clock rate, CPI 영향을 끼친다.)

Power Trend



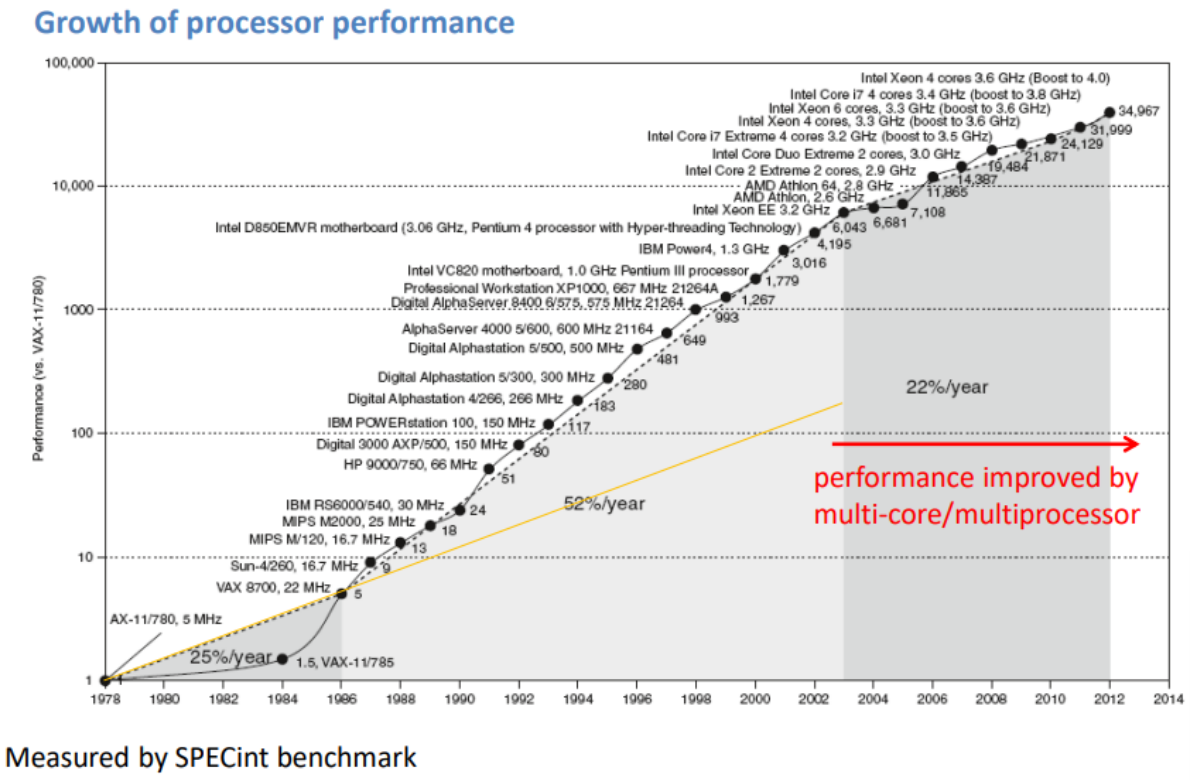
$$\text{Power} \propto \text{Capacitive load} \times \text{Voltage}^2 \times \text{Frequency}$$

1. 더 빠른 clock rate는 더 나은 성능을 의미한다.
2. 과거에는 빠른 속도로 clock rate와 power가 증가해 왔지만, 현재에 이르서는 clock rate와 power의 상승 폭이 점점 낮아지고 있다.
    a. 이는 power가 높을 수록 발열이 심해서져 cooling 문제가 나타나기 때문이다.
3. 필요한 전압을 줄이면 clock 속도를 높일 수 있다.

    → 하지만, 전압을 줄이는 것에도 한계가 있다.

- 그렇다면 수행을 향상시키기 위해서는 어떤 것을 해야할까?

From Uniprocessor to Multiprocessor



Measured by SPECint benchmark

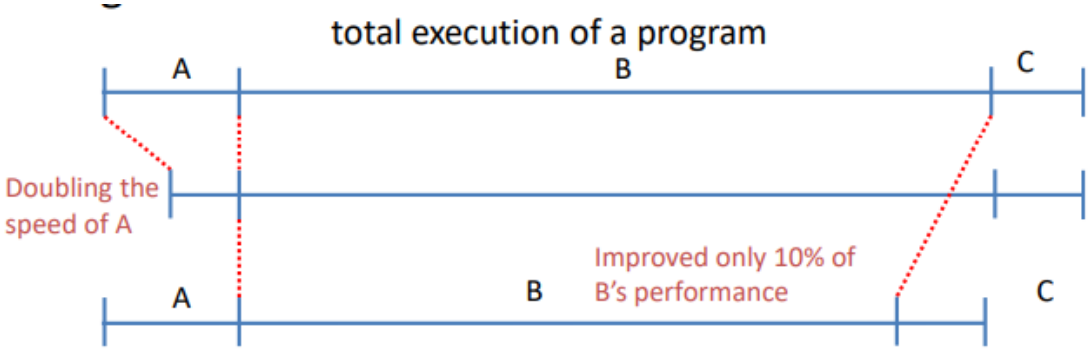
Implication

- Previously
  - 멀티 코어 이전에는 하드웨어 성능이 증가하면, 소프트웨어 성능도 자동적으로 좋아졌다. 어떠한 변화없이도.
- Multi-core(=processor) H/W
  - dual-core, quad-core … etc
  - 멀티코어를 활용하기 위해서는 S/W를 다시 작성해야 한다.
    - 병렬 프로그래밍의 어려움
      - load balancing
      - communication overhead
      - synchronization
- 멀티 코어가 등장 한 후 , 소프트웨어의 성능을 위해선 소프트웨어가 멀티 코어를 잘 사용하도록(즉, 병렬 컴퓨팅이 가능하도록) 개발하는 것이 필수가 되어버렸다.
- 하지만, 병렬 프로그래밍은 매우 어려운 일인데, 어떻게 코어들에게 일을 분배할지 고민해야 하고, 코어들끼리 소통을 하면서 오버 헤드가 발생하기도 하기에 이를 방지하는 프로그래밍을 구성해야 하며, 동기화 문제도 발생한다.

- 그래서 멀티 코어가 등장한 이후, 현재는 하드웨어 성능을 높이기 위해 노력하기 보다는 소프트웨어 성능을 높이기 위해 많은 노력을 하고 있다.
- 위 그래프를 참고해보면, 멀티 코어가 등장한 이후 성장 폭이 22%로 감소했다. 이는 암달(Amdahl's)의 법칙을 통해 설명이 가능하다고 한다.

Amdalh's Low

- Amdalh 의 법칙은 프로그램의 일부를 개선할 때, 전체적으로 얼만큼의 최대 성능 향상이 있는지 계산하는 데 사용된다.



- Common misbelief
  - 흔히들, 프로그램의 한 부분을 향상시킨 만큼 전체적인 프로그램의 성능도 같은 만큼 향상된다고 생각하는 것이다. 하지만, 이는 틀린 말인데, 예를 들어, 위 그림에서 segment A의 속도를 50% 향상한 것과 segment B의 속도를 10% 향상한 것이 전체 프로그램의 속도는 같다. 즉, 프로그램의 전반적인 향상을 위해선 segment B의 속도를 높이는 것이 A를 높이는 것보다 더 많은 영향을 미친다는 뜻이다.
- 또한, Amdalh's Low에서 여러 개의 프로세서를 사용하는 병렬 컴퓨팅에서 프로그램의 속도가 불가능한 작업들에 의해 제한된다. 이는 일부 병렬화가 가능한 작업들은 계산에 참여하는 프로세서 수에 비례해 속도가 빨라지지만, 병렬화가 불가능한 작업은 빨라지지 않기 때문이다.
- 즉, 프로세서가 아무리 많아져도 속도는 한계가 정해져 있다는 소리다. 이로 인해 Amdalh's Low의 법칙은 Amdalh's 의 저주라는 별명이 붙게 됐다
- 예제를 보자, 어떤 프로그램이 실행하는데 100초가 걸린다. 그 중 곱하기 연산이 80초를 소모한다고 치자. 나는 이 프로그램을 5배 빠르게 하고 싶다. 그렇다면, 곱하기 연산은 얼마 만큼 빠르게 해야 할까?

- Execution time after improvement

$$= \frac{Execution\ time\ affected\ by\ improvement}{Amount\ of\ improvement} + Execution\ time\ unaffected$$

T = 80/n + (100-80)

5 times faster = 20 seconds

20 = 80/n + 20

즉, 80/n = 0

n 은 존재하지 않는다. 따라서 곱하기 연산을 아무리 빠르게 하고싶어도 전체 프로그램의 속도를 5배 빠르게 할 수 없다.