Computer System Overview

'20H2

송 인 식

Outline

- How Computers Work
- Abstractions
- Evolution of Computers
- Design Principles

Information is Bits + Context

```
#include <stdio.h>

int main()
{
    printf("hello, world\n");
    return 0;
}
```

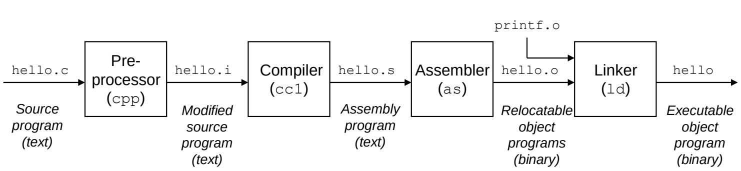
code/intro/hello.c

Information is Bits + Context

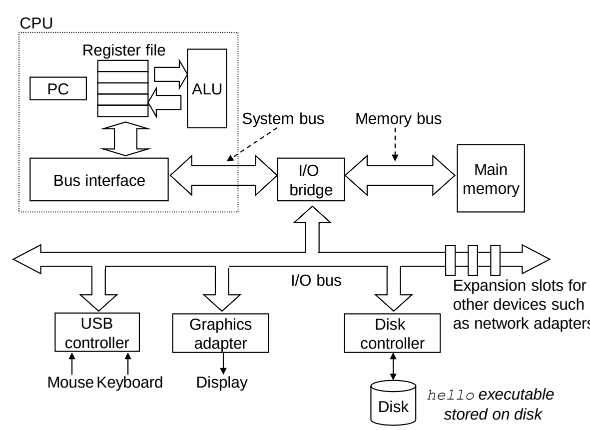
#	i	n	c	l	u	d	e	*SP*	<	s	t	d	i	o	.
35	105	110	99	108	117	100	101	32	60	115	116	100	105	111	46
h	>	\n	\n	i	n	t	*SP*	m	a	i	n	(	)	\n	{
104	62	10	10	105	110	116	32	109	97	105	110	40	41	10	123
\n	*SP*	*SP*	*SP*	*SP*	p	r	i	n	t	f	(	"	h	e	l
10	32	32	32	32	112	114	105	110	116	102	40	34	104	101	108
l	o	,	*SP*	w	o	r	l	d	\	n	"	)	;	\n	*SP*
108	111	44	32	119	111	114	108	100	92	110	34	41	59	10	32
SP	*SP*	*SP*	r	e	t	u	r	n	*SP*	0	;	\n	}	\n	
32	32	32	114	101	116	117	114	110	32	48	59	10	125	10	

The ASCII text representation of hello.c.

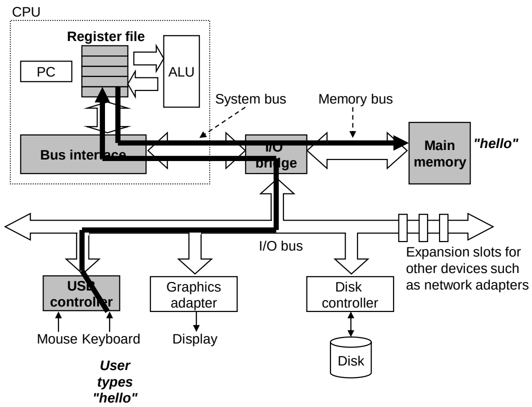
The Compilation System



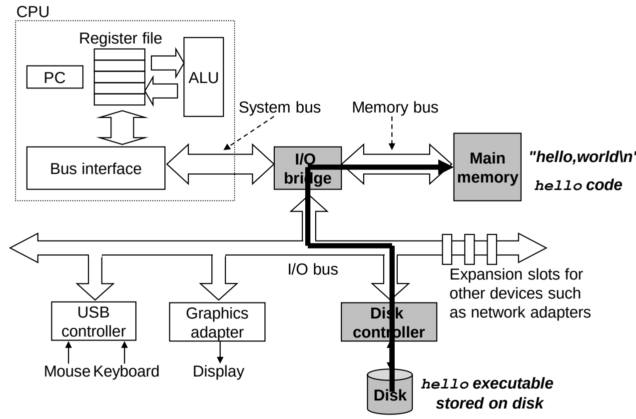
Hardware Organization of a Typical System



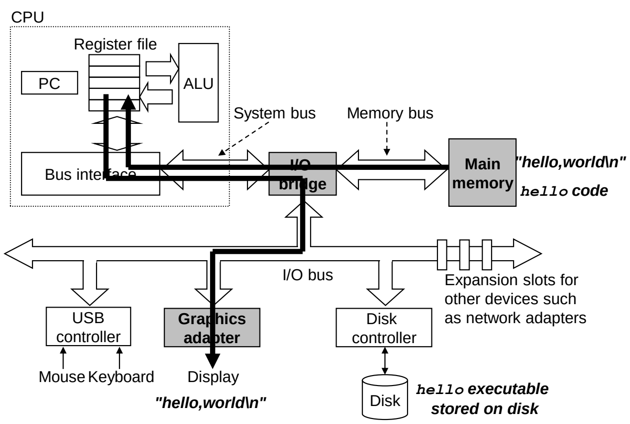
Reading the hello command from the keyboard



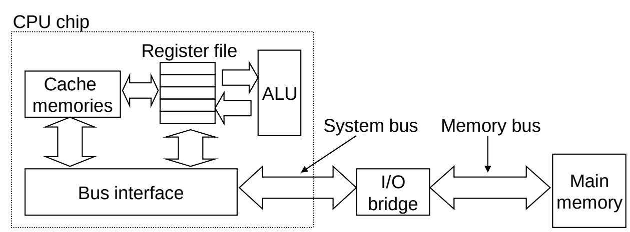
Loading the executable from disk into main memory



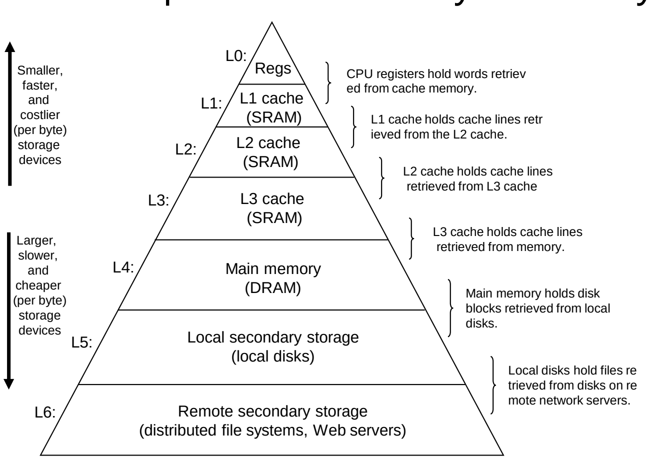
Writing the output string from memory to the display



Cache memories



An example of a memory hierarchy

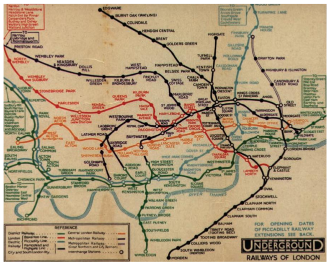


Outline

- How Computers Work
- Abstractions
- Evolution of Computers
- Design Principles

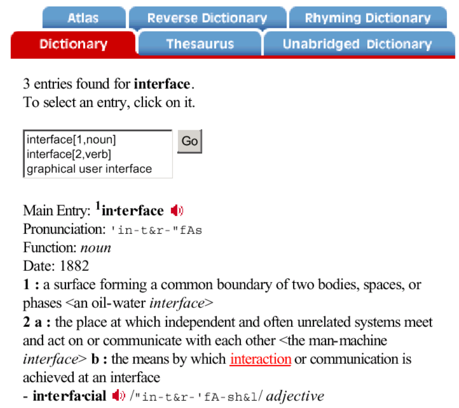
Abstraction

- Before After



Jeff Kramer, "Is Abstraction the Key to Computing," Communications of ACM, April 2007, Vol. 50, No. 4, pp. 37 - 42.

Interface



Atlas | Reverse Dictionary | Rhyming Dictionary
Dictionary | Thesaurus | Unabridged Dictionary

3 entries found for **interface**.
To select an entry, click on it.

interface[1,noun]
interface[2,verb]
graphical user interface
Go

Main Entry: **[1]in·ter·face**
Pronunciation: 'in-t&r-"fAs
Function: *noun*
Date: 1882
1 : a surface forming a common boundary of two bodies, spaces, or phases <an oil-water *interface*>
2 a : the place at which independent and often unrelated systems meet and act on or communicate with each other <the man-machine *interface*> **b :** the means by which interaction or communication is achieved at an interface
- **in·ter·fa·cial** /"in-t&r-'fA-sh&l/ *adjective*

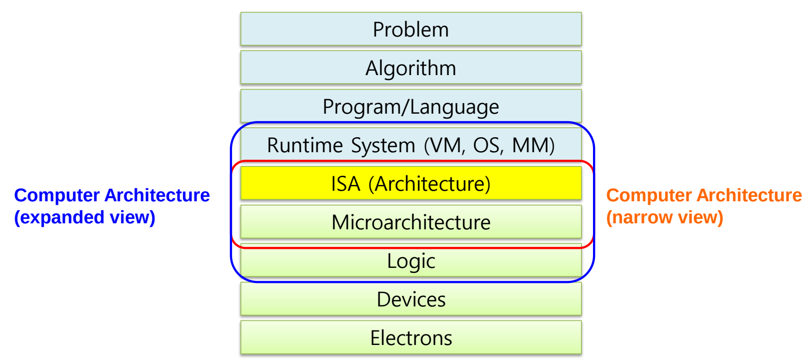
The HW/SW Interface

Application software	a[i] = b[i] + c;
	↓ Compiler

Systems software (OS, compiler)	lw $15, 0($2)
	add $16, $15, $14
	add $17, $15, $13
	lw $18, 0($12)
	lw $19, 0($17)
	add $20, $18, $19
	sw $20, 0($16)

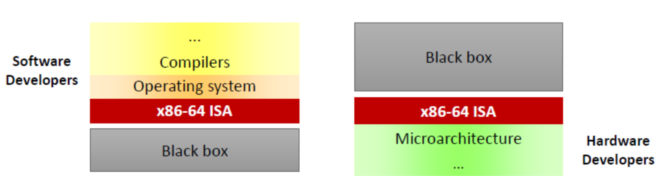
	↓ Assembler
Hardware	000000101100000 110100000100010 …

Levels of Abstractions



Instruction Set Architecture

- The hardware/software interface
  - Hardware abstraction visible to software (OS, compilers, …)
  - Instructions and their encodings, registers, data types, addressing modes, etc.
  - Written documents about how the CPU behaves
  - e.g. All 64-bit Intel CPUs follow the same x86-64 (or Intel 64) ISA



Levels of Program Code

- High-level language
  - Level of abstraction closer to problem domain
  - Provides for productivity and portability
- Assembly language
  - Textual representation of instructions
  - For humans
- Machine language
  - Hardware representation
  - Binary digits (bits)
  - Encoded instructions and data

Abstraction is Good, But …

- Abstraction helps us deal with complexity
  - Hide lower-level details
  - E.g. Abstract data types, Asymptotic analysis
- These abstractions have limits
  - Especially in the presence of bugs
  - Need to understand details of underlying implementations
- This is why you should take this course seriously even if you don't want to be a computer architect!

Great Reality #1: Int's are not Integers, Float's are not Reals

- Example 1: Is $x^2 \geq 0$?
  - Float's: Yes!
  - Int's:
    - 40000 * 40000 → 1600000000
    - 50000 * 50000 → ??
- Example 2: Is (x + y) + z = x + (y + z)?
  - Unsigned & Signed Int's: Yes!
  - Float's:
    - (1e20 + -1e20) + 3.14 → 3.14
    - 1e20 + (-1e20 + 3.14) → ??

Computer Arithmetic

- Does not generate random values
  - Arithmetic operations have important mathematical properties
- Cannot assume all "usual" mathematical properties
  - Due to finiteness of representations
  - Integer operations satisfy "ring" properties
    - Commutativity, associativity, distributivity
  - Floating point operations satisfy "ordering" properties
    - Monotonicity, values of signs
- Observation
  - Need to understand which abstractions apply in which contexts
  - Important issues for compiler writers and serious application programmers

Great Reality #2: You've Got to Know Assembly

- Chances are, you'll never write programs in assembly
  - Compilers are much better & more patient than you are
- But: Understanding assembly is key to machine-level execution model
  - Behavior of programs in presence of bugs
    - High-level language models break down
  - Tuning program performance
    - Understand optimizations done / not done by the compiler
    - Understanding sources of program inefficiency
  - Implementing system software
    - Compiler has machine code as target
    - Operating systems must manage process state
  - Creating / fighting malware
    - x86 assembly is the language of choice!

Great Reality #3: Memory Matters Random Access Memory Is an Unphysical Abstraction

- Memory is not unbounded
  - It must be allocated and managed
  - Many applications are memory dominated
- Memory referencing bugs especially pernicious
  - Effects are distant in both time and space
- Memory performance is not uniform
  - Cache and virtual memory effects can greatly affect program performance
  - Adapting program to characteristics of memory system can lead to major speed improvements

Memory Referencing Bug Example

```
typedef struct {
  int a[2];
  double d;
} struct_t;

double fun(int i) {
  volatile struct_t s;
  s.d = 3.14;
  s.a[i] = 1073741824; /* Possibly out of bounds */
  return s.d;
}
```

```
fun(0) -->     3.14
fun(1) -->     3.14
fun(2) -->     3.1399998664856
fun(3) -->     2.00000061035156
fun(4) -->     3.14
fun(6) -->     Segmentation fault
```

- Result is system specific

Memory Referencing Bug Example

```
typedef struct {
  int a[2];
  double d;
} struct_t;
```

```
fun(0) -->    3.14
fun(1) -->    3.14
fun(2) -->    3.1399998664856
fun(3) -->    2.00000061035156
fun(4) -->    3.14
fun(6) -->    Segmentation fault
```

Explanation:

	Critical State	6
	?	5
	?	4
struct_t	`d7 ... d4`	3
struct_t	`d3 ... d0`	2
struct_t	`a[1]`	1
struct_t	`a[0]`	0

Location accessed by `fun(i)`

Memory Referencing Errors

- C and C++ do not provide any memory protection
  - Out of bounds array references
  - Invalid pointer values
  - Abuses of malloc/free
- Can lead to nasty bugs
  - Whether or not bug has any effect depends on system and compiler
  - Action at a distance
    - Corrupted object logically unrelated to one being accessed
    - Effect of bug may be first observed long after it is generated
- How can I deal with this?
  - Program in Java, Ruby, Python, ML, …
  - Understand what possible interactions may occur
  - Use or develop tools to detect referencing errors (e.g. Valgrind)

Great Reality #4: There's more to performance than asymptotic complexity

- Constant factors matter too!
- And even exact op count does not predict performance
  - Easily see 10:1 performance range depending on how code written
  - Must optimize at multiple levels: algorithm, data representations, procedures, and loops
- Must understand system to optimize performance
  - How programs compiled and executed
  - How to measure program performance and identify bottlenecks
  - How to improve performance without destroying code modularity and generality

Memory System Performance Example

```
void copyij(int src[2048][2048],
            int dst[2048][2048])
{
  int i,j;
  for (i = 0; i < 2048; i++)
    for (j = 0; j < 2048; j++)
      dst[i][j] = src[i][j];
}
```

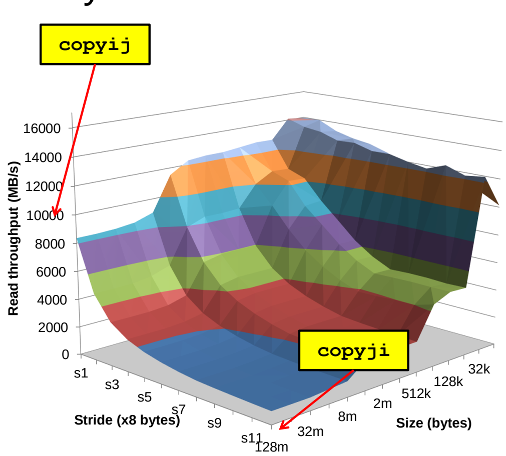
```
void copyji(int src[2048][2048],
            int dst[2048][2048])
{
  int i,j;
  for (j = 0; j < 2048; j++)
    for (i = 0; i < 2048; i++)
      dst[i][j] = src[i][j];
}
```

4.3ms   81.8ms

2.0 GHz Intel Core i7 Haswell

- Hierarchical memory organization
- Performance depends on access patterns
  - Including how step through multi-dimensional array

Why The Performance Differs



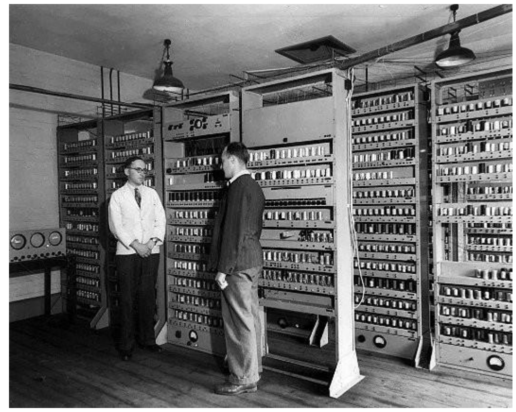
Great Reality #5: Computers do more than execute programs

- They need to get data in and out
  - I/O system critical to program reliability and performance
- They communicate with each other over networks
  - Many system-level issues arise in presence of network
    - Concurrent operations by autonomous processes
    - Coping with unreliable media
    - Cross platform compatibility
    - Complex performance issues

Outline

- How Computers Work
- Abstractions
- Evolution of Computers
- Design Principles

The 1st Generation Computer

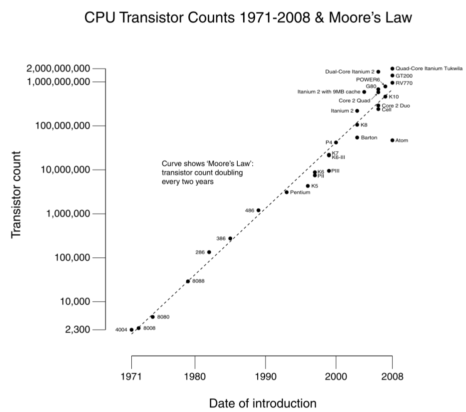


Source: http://www.computerhistory.org

The Computer Revolution

- Progress in computer technology
  - Underpinned by Moore's Law
- Makes novel applications feasible
  - World Wide Web (WWW)
  - Smartphones
  - Search engines
  - Human genome project
  - Self-driving cars
  - Artificial intelligence
  - VR/AR
- Computers are pervasive

Moore's Law

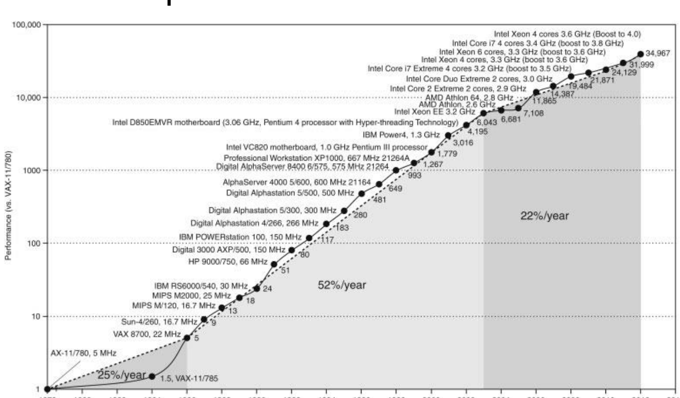


"The number of transistors incorporated in a chip will approximately double every 24 months."

Gordon Moore, Intel Co-founder

(1965)

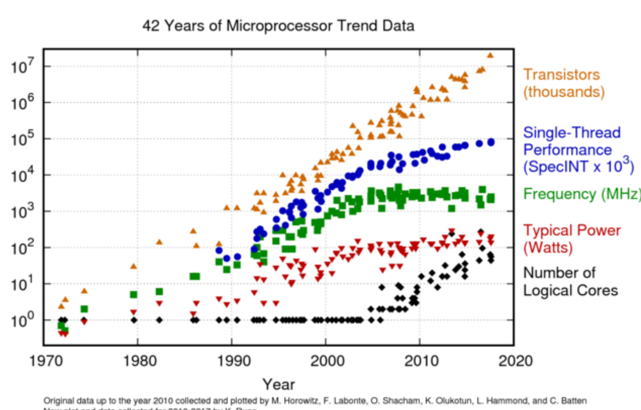
Microprocessor Performance



50% improvement every year!!
What contributes to this improvement?

Source: H&P

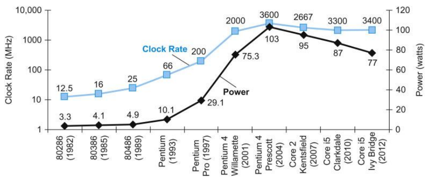
Microprocessor Performance



Source: karlrupp.net

Power Consumption Trends

- Dyn. power ∝ activity x capacitance x voltage^2 x frequency
- Voltage and frequency are somewhat constant now, while capacitance per transistor is decreasing and number of transistors (activity) is increasing
- Leakage power is also rising (function of #trans and voltage)

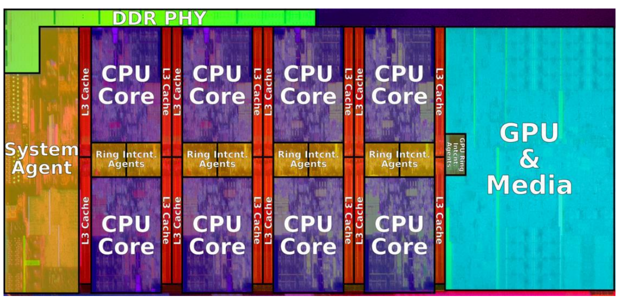


Source: H&P

Important Trends

- Running out of ideas to improve single thread performance
- Power wall makes it harder to add complex features
- Power wall makes it harder to increase frequency
- Additional performance provided by: more cores, occasional spikes in frequency, accelerators

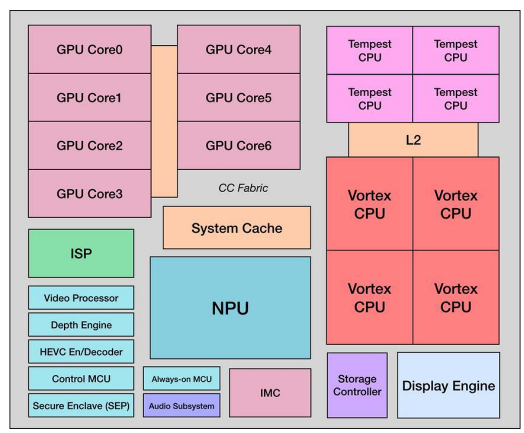
Intel Core i9-9900K (Coffee Lake, 2018)



Process:
14nm
Transistors:
~ 3B
Die size:
~ 177 mm²

Source: https://en.wikichip.org/wiki/intel/core_i9/i9-9900k

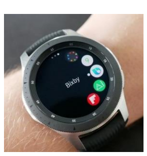
Apple A12X Bionic (2018)



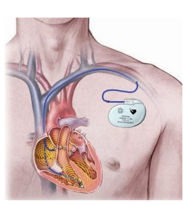
Process:
7nm
Transistors:
~ 10B
Die size:
~ 122 mm²

Source: https://en.wikichip.org/wiki/apple/ax/a12x

Computers Today






Source: http://www.computerhistory.org

Outline

- How Computers Work
- Abstractions
- Evolution of Computers
- Design Principles

Classes of Computers

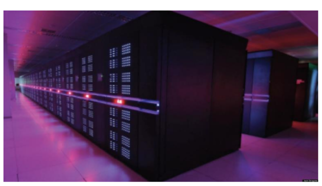
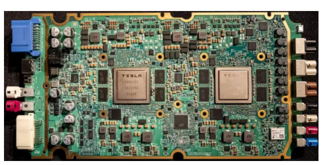
- Personal computers
  - General purpose, variety of software
  - Subject to cost/performance tradeoff
- Server computers
  - Network based
  - High capacity, performance, reliability
  - Range from small servers to large data centers
- Supercomputers
  - High-end scientific and engineering calculations
  - Highest capability but represent a small fraction of the overall computer market
- Embedded computers
  - Hidden as components of systems
  - Stringent power/performance/cost constraints

Different Platforms, Different Goals



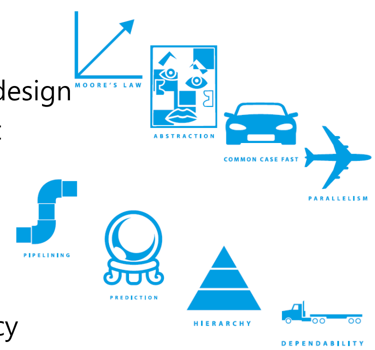

Different Platforms, Different Goals



Eight Great Ideas in Computer Architecture

- Design for Moore's Law
- Use abstraction to simplify design
- Make the common case fast
- Performance via parallelism
- Performance via pipelining
- Performance via prediction
- Hierarchy of memories
- Dependability via redundancy



Questions?