UNIT-4 COMPUTER ABSTRACTIONS & TECHNOLOGIA

Translating and Starting a Program

→ A modern computer doesn't run C code directly.

A program goes through 4 major translation stages before it can run:

Compiler
↓
Assembler
↓
Linker
↓
Loader

→ **Compiler**

→ The role of compiler is to translate high level language to Assembly level Programs

→ Assume input to be C or C++, then output would be RISC-V assembly

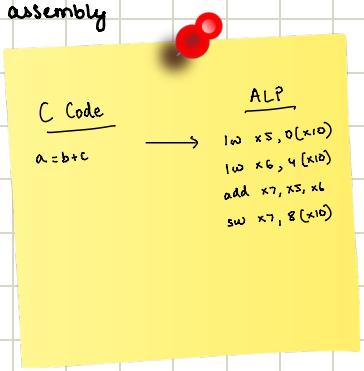
→ Functions of Compiler

→ Translate high level constructs (loops, if-else) into RISC-V instructions

→ Assign registers

→ Break code into basic blocks

→ Perform Optimizations



→ **Assembler**

→ The assembler takes input as assembly level program and gives output as an object file which contains machine code, symbol table, relocation entries

→ They assign addresses relative to start of each section

→ Pseudo instructions give RISC-V a richer set of assembly instructions & assembler translates pseudo instructions into actual instructions

→ It also creates symbol table which consists of Function names, Global Variables, Labels

→ It creates relocation entries for the addresses not known yet (like la, branch etc.,)

→ But assembler can't resolve thing that are defined in another file, this is where linker comes in

```
.text     → machine instructions
.data     → global/static variables
.rodata   → constants
.symtab   → symbol information
.reloc    → relocation entries
```

⟶ Object file selections

→ **Linker**

→ It combines multiple object files to form a single executable file (ELF)

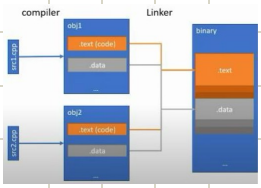
→ **Linker Responsibilities**

→ Resolve symbol references (ex: file A calls a function in file B) (Linker matches symbols to definitions)

→ Resolve library functions (ex: printf, scanf, malloc etc.,)

→ Place code & data in final memory layout

text	code
rodata	constants
data	initialized globals
bss	uninitialized globals



→ Perform relocation (Fix addresses in instructions like jal, beq, loads of addresses, global variable access)

→ Produce final executable (ELF for linux, .exe for windows)

→ **Loader**

→ It is a part of OS

→ **Loader Functions :**

→ Loads text & data sections into memory

→ Allocates space for heap & stack

→ Set register values (sp - top of stack, pc - entry point of _start)

→ Set environment variables

→ Jump to program's _start (basically calling main())

→ After main() returns, it cleans up & returns to OS, using exit system call

→ **Statically Linked Libraries**

→ A library where the actual library code is copied into the executable during linking stage

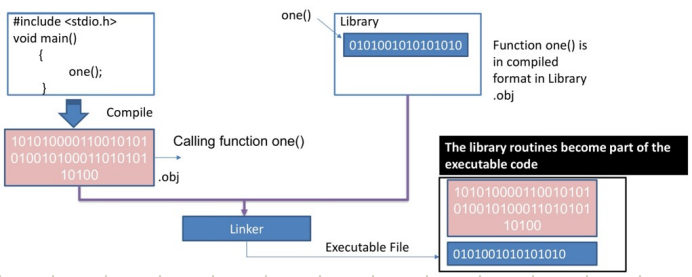
ex: libc.a, libm.a, .a

→ The linker only takes required func's & embeds them into the final program

→ The program contains everything & doesn't require to download any other libraries during run time

→ It's executable is self-contained, faster execution startup, and stable

→ But, executable's size is too large, increase RAM usage & really hard to update



→ **Dynamically Linked Libraries**

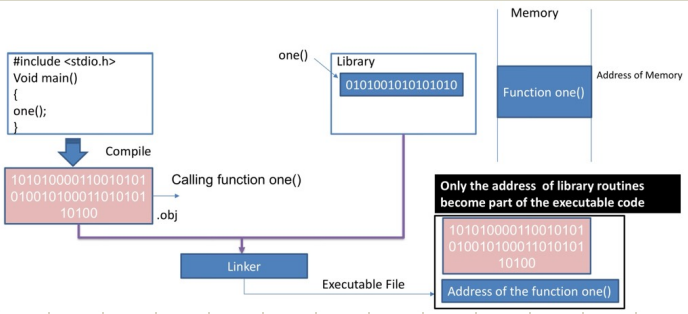
→ Instead of copying into executable, it only stores the reference to libraries

→ Actual library code is stored in .so (Linux) & .dll (windows)

→ When running the program, loader loads shared libraries into memory and links to program, fixes the addresses and connects function calls dynamically

→ It has very small executable & RAM efficient so libraries can be updated w/o recompiling programs

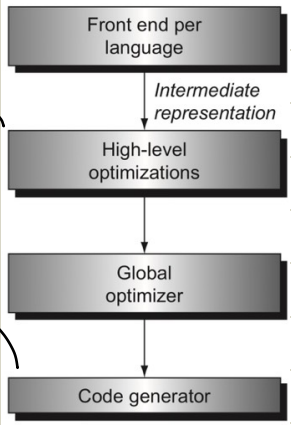
→ But it is slow, and program fails if shared library is missing & some older versions of the program may face version compatibility issues



Compiling C

→ Modern compilers are built as multi-pass pipelines where each pass performs one type of analysis or optimization & passes the output to next pass

→ To achieve this, compiler is broken into 4 passes / phases :



Reads C program,
Produces intermediate representation (IR) *
Handles language-specific parts
Does parsing & type-checking

Inlining (Procedure integration)
Loop Transformations
Work on IR (machine-independant)

Works across basic blocks
Performs global common subexpr elimination,
constant propagation & dead code elimination
Performs global register allocation

Converts IR → Actual RISC-V instructions
Instruction selection
Instruction scheduling
Machine-dependant optimizations

→ Now let's understand each phase

→ **Front End**

→ This is the language dependant part of the compiler

→ Its job is to understand source code, check for correctness & convert to clean format (IR) for backend optimizing

→ Front-end performs 4 functions :

i) **Scanning** (Lexical Analysis)

→ It reads the raw characters and then makes tokens

ex: while (save[i] == k) i += 1 ;

Token sequence → while , (, save , [, i ,] , == , k ,) , i , += , 1

→ We do this because it is impossible for the parser to work directly on characters like w , h , i , l , e , (, s , a , v , e

So, token gives structure

ii) **Parsing** (Syntax Analysis)

→ It ensures that the structure of the code matches C grammar rules

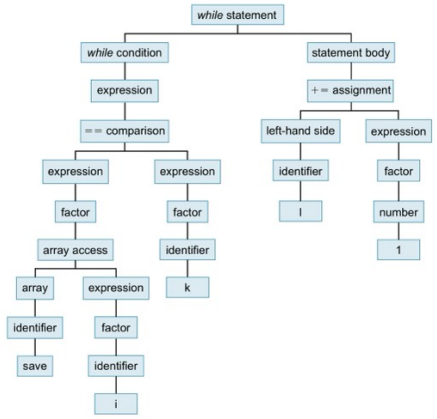
→ It takes input as token stream & gives output as abstract syntax tree (AST)

→ AST is required because token alone doesn't show structure

ex: Let token stream be i , + , j , * , k

This means it could be (i+j)*k or i+(j*k)

Only parsing ensures correct grouping based on procedure



iii) Semantic Analysis

→ It is used to check meaning as well as correctness of the program

→ This includes :

i) Type checking (Ensure operation match types)

```
int i;
i = i + 3;        ✓
i = i + "hello"; ×
```

ii) Declaration checking (Ensure variables are declared before use)

```
x = 3;   ×  error if x isn't declared
```

iii) Scope checking (Ensure variable reference are legal based on scope)

iv) Build the SYMBOL TABLE

→ Stores variable names, types, memory locations, func^{n} parameters, array sizes, struct definitions

v) Detect Semantic Errors

ex: passing wrong no. of arguments, assigning float to int w/o cast, using uninitialized variables

→ Finally output is fully checked AST, Symbol Table, Type information for every expression

iv) IR Generation

→ Translate the AST + symbol table into machine independant, low-level instruction set

→ This representation looks like RISC-V assembly but uses ∞ virtual registers (r100, r101 etc.,)

ex: while (save[i] == k)
    i += 1;

→

```
loop:
  # comments are written like this--source code often included
  # while (save[i] == k)
  addi r100, x0, save          # r100 = &save[0]
  lw   r101, i
  addi r102, x0, 4
  mul  r103, r101, r102
  add  r104, r103, r100
  lw   r105, 0(r104)           # r105 = save[i]
  lw   r106, k
  bne  r105, r106, exit
  # i += 1
  lw   r106, i
  addi r107, r106, i           # increment
  sw   r107, i
  jal x0 loop                  # next iteration
exit:
```

→ IR Properties :

→ No real registers yet

→ Each tempory gets its own virtual register

→ Easier to optimize than real assembly

→ Machine independant (same IR for RISC-V/ARM/x86 back ends)

→ Compiler priorities :

→ Produce correct code

→ Produce fast code

→ Produce minimal code size

→ **High-Level Optimizations**

→ They are transformations applied early in the compiler, right after front-end generates IR

→ They are called high-level because they operate at a level close to source code than machine level

→ These optimizations

- → Improve program performance
- → Reduce no. of operations
- → Make loops faster
- → Enable more aggressive global or local optimizations later

→ There are 3 major classes :

i) **Procedure Inlining**

→ Inlining replaces a function call with the body of the function

ex :
```
int square (int x) {
    return x * x; }

y = square (a);
```
After Inlining → `y = a * a`

→ Advantages :

i) Removes function call overhead

ii) Improves performance

iii) Exposes more opportunities for other optimizations

→ Disadvantages :

i) Increase code size

ii) Excessive inling can reduce performance

ii) **Loop Optimizations**

→ Loops run many times, so we optimize gives massive performance gain

→ Several key loop transformations exist, few include :

i) **Loop Unrolling**

→ Replicate loop body multiple times to reduce loop overhead

ex :
```
for (i=0; i<4; i++)
    sum += a[i];
```
unrolled →
```
sum += a[0]
sum += a[1]
sum += a[2]
sum += a[3]
```

→ Fewer branch instr^n, Better pipeline efficiency, Better use of registers

ii) **Loop Interchange** - Reorders nested loops for better memory access

iii) **Loop Blocking** - Break loops into smaller blocks that fit into cache

iv) **Loop Fusion & Fission** - Fusion → Combine 2 loops into 1

Fission → Break 1 loop into 2

v) **Loop-Invariant Code Motion** - Moves calculations that don't change inside the loop, outside it

→ **Local and Global Optimizations**

→ After Front-end & High level optimizations, compiler enters the local & global optimization phase

→ This phase contains 3 classes of optimizations

i) Local Optimization - Within single basic block

ii) Global Optimization - Across basic blocks

iii) Global register allocation - Across a function

→ **Local Optimization**

→ A basic block is a straight-line sequence of code with one entry, one exit & no branches inside

→ It is easy because no control flow changes, no need to track variables across branches & no ambuigity about which statements to execute

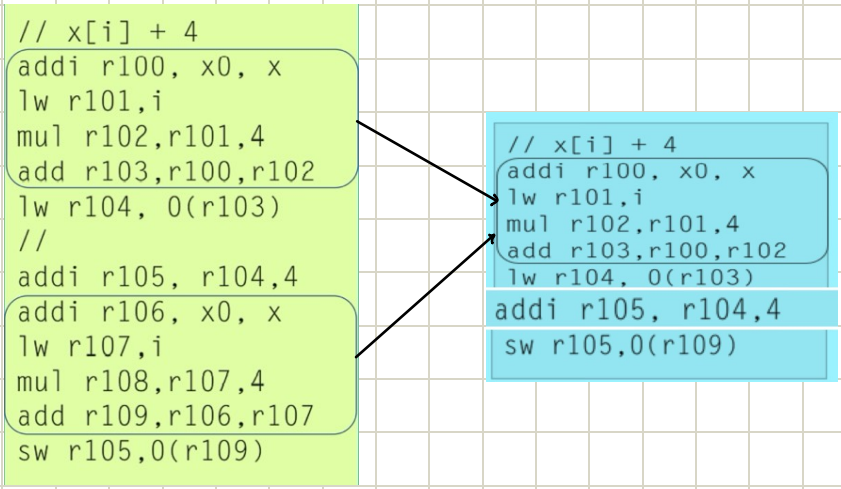
→ Local optimizations run before & after global optimizations & during code generation
(before: To clean IR; after: To remove leftovers; during: peephole)

→ **Techniques**

1) Local Common Subexpression Elimination (cse)

→ If the same expression occurs twice, it replaces 2nd one by reference to the first

ex:
```
a = b*c + g        Optimized as →   temp = b*c
d = b*c + K                         a = temp + g
                                    d = temp + g
```



2) Strength Reduction

→ Replace expensive operations with cheaper ones, like,

mul r, x, 4 ⟶ slli r, x, 2

division ⟶ shifts

multiplication in powers of 2 ⟶ shifts

→ Hardware complexity of multiplier is more than shifter as a result

3) Constant Propagation: Find constants in code & propagate them, collapsing values whenever possible

a = 5

b = a + 3 ⟶ b = 8

Constant Folding: Evaluation of an expression with constant operand to replace expression with value

3 + 4 ⟶ 7

2 * 3 ⟶ 6

PI = 22/7 ⟶ PI = 3.14

4) Dead Store Elimination

→ Finds stores to values that aren't used again and eliminates the store

```
c = a*b          optimize        c = a*b
t = a            ------->        d = a*b + 4
d = a*b + 4
```

Dead code Elimination

→ Finds unused code - code that can't affect the result of the program - and eliminates it.

```
// x[i] + 4
addi r100, x0, x
lw r101,i
mul r102,r101,4
add r103,r100,r102
lw r104, 0(r103)
//
addi r105, r104,4
addi r106, x0, x
lw r107,i
mul r108,r107,4
add r109,r106,r107
sw r105,0(r109)
```

```
// x[i] + 4
addi r100, x0, x
lw r101,i
slli r102,r101,2
add r103,r100,r102
lw r104, 0(r103)
// value of x[i] is in r104
addi r105, r104,4
sw r105, 0(r103)
```

5) Peephole Optimization

→ Inspect a small window of instructions:

i) Remove redundant moves

ii) Collapse instruction sequences

iii) Detect unnecessary loads/stores

iv) Combine instructions

ex: `addi x5, x5, 0` → Remove

`lw x5, 0(x5)` → Remove

`mv x3, x3` → Remove

→ Global Optimization

→ Now, the compiler must look throughout the entire function, not just one block

→ Two other important global optimizations:

1) Code Motion

→ Finds code that is invariant to loops & computes same value on every iteration

```
while ( i<100)
   {
    a= sin(x)/cos(x) + i;
    i++;
  }
```

Optimized to →

```
K= sin(x)/cos(x)
while (i<100) {
a= K+i; i++; }
```

2) Induction Variable Elimination (IVE)

→ A variable that changes every loop is known as induction variable & it is iterated by some constant. These variables have some relation which can be used to replace them.

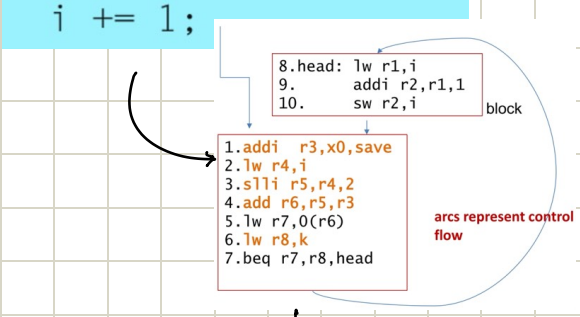
```
int max = 10;
int result = 0;
for (int i = 0; i < max; i++) {
    result += 2;
}
return result;
```

→

```
int max = 10;
int result = 0;
for (; result < max*2; result+=2) {}
return result;
```

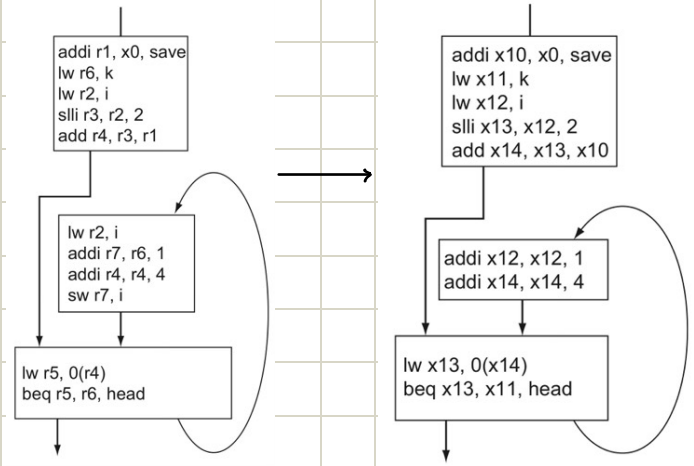
↳ eliminate "i" by replacing its uses with another basic induction variable "result"

→
```
while (save[i] == k)
  i += 1;
```



→ Important transformation performed was to move the while test & unconditional branch to the end

This eliminates the unconditional branch that was formerly inside the loop & places it before loop



No. of instructions = 6

Before register allocation

No. of instructions = 4

After register allocation

→ The control flow graph showing the representation of while loop before and after Register allocation

→ No. of instructions in inner loop reduced from 6 to 4

→ Register allocation makes sure that in a region / block registers, 2 variables DO NOT share the same register otherwise, compiler overwrites one value when writing the other leading to a race between registers.

→ This entire process of Global Optimization is hard because compiler must be conservative & it can't apply an optimization unless it is 100% safe to all inputs

But there are some barriers:

→ Aliasing (2 pointers may refer to same memory)

→ Unknown function side effects

→ Branches that change control flow

→ Loop variables changed inside loop

→ **Code Generation**

→ The final steps of the compiler are code generation and assembly

→ In modern compilers, this compilation process uses pattern matching with either a tree-based pattern matcher or a pattern matcher driven by a parser.

→ During code generation, the final stages of machine-dependant optimization are also performed.

→ Major Types of Optimization & explanation of each class

Optimization name	Explanation	gcc level
High level	***At or near the source level; processor independent***	
Procedure integration	Replace procedure call by procedure body	03
Local	***Within straight-line code***	
Common subexpression elimination	Replace two instances of the same computation by single copy	01
Constant propagation	Replace all instances of a variable that is assigned a constant with the constant	01
Stack height reduction	Rearrange expression tree to minimize resources needed for expression evaluation	01
Global	***Across a branch***	
Global common subexpression elimination	Same as local, but this version crosses branches	02
Copy propagation	Replace all instances of a variable *A* that has been assigned *X* (i.e., *A* = *X*) with *X*	02
Code motion	Remove code from a loop that computes the same value each iteration of the loop	02
Induction variable elimination	Simplify/eliminate array addressing calculations within loops	02
Processor dependent	***Depends on processor knowledge***	
Strength reduction	Many examples; replace multiply by a constant with shifts	01
Pipeline scheduling	Reorder instructions to improve pipeline performance	01
Branch offset optimization	Choose the shortest branch displacement that reaches target	01

Arithmetic for Computers (Addition & Subtraction)

→ We know that negative numbers are stored in two's complement form & Range for two's complement is -2^{N-1} to $2^{N-1} - 1$

if 32 bit, -2147483648 to 2147483647

→ Overflow occurs where adding 2 positive numbers and sum is negative (or)
adding 2 negative numbers and sum is positive
subtracting negative from positive and sum is negative
subtracting positive from negative and sum is positive

No overflow can occur when adding positive and negative operands

Operation	Operand A	Operand B	Result indicating overflow
$A + B$	≥ 0	≥ 0	< 0
$A + B$	< 0	< 0	≥ 0
$A - B$	≥ 0	< 0	< 0
$A - B$	< 0	≥ 0	≥ 0

→ Overflow with unsigned integers

→ Unsigned integers are commonly used for memory addresses where overflows are ignored

→ Fortunately, compiler can easily check for unsigned overflow using branch instruction

→ Addition has overflowed if sum is less than either of the addends
Subtraction has overflowed if difference is greater than the minuend

→ So, in some programming languages, we can declare integers of different sizes, hence, a program can perform arithmetic on 2's complement numbers.
For 8-bit value → Byte arithmetic
For 16-bit value → Half arithmetic

→ But in the case of RISC-V, it doesn't support 8-bit & 16-bit arithmetic instructions
All of them are 32 bit (word) operations

→ So we can avoid this issue using a loophole :
i) Load with lb, lh into 32-bit registers
ii) Perform arithmetic with add, sub, mul, div (Even if 8-bit, we use 32-bit arithmetic)
iii) Store with sb, sh
This automatically truncates extra bits

→ Saturation Arithmetic

→ One feature generally not found in general-purpose microprocessors is saturating operations

→ Saturation is when calculation overflows, result is set to largest positive number or most negative number, rather than modulo calculation / two's complement arithmetic

1) What is saturation arithmetic?
   a) Saturation arithmetic means that when a calculation overflows, the result is set to the largest positive number or the most negative number
   b) Saturation arithmetic means a modulo calculation as in two's complement arithmetic.
   c) Saturation arithmetic means that when a calculation overflows, the result is set zero
   d) Saturation arithmetic means that when a calculation overflows, the result is set to the largest positive number

2) In the case of signed numbers, which of the following condition(s) is an overflow condition for subtraction?
   a) Overflow occurs in subtraction when we subtract a negative number from a positive number and get a negative result.
   b) Overflow occurs in subtraction when we subtract a positive number from a negative number and get a positive result.
   c) Overflow occurs in subtraction when we subtract a negative number from a positive number and get a negative result, or when we subtract a positive number from a negative number and get a positive result.
   d) None of these

4) In the case of unsigned numbers, Which condition indicates the overflow condition for an addition?
   a) Addition has overflowed if the sum is less than both of the addends
   b) Addition has overflowed if the sum is greater than either of the addends
   c) Addition has overflowed if the sum is less than either of the **addends**
   d) None of these

5) In the case of unsigned numbers, Which condition indicates the overflow condition for subtraction?
   a) If the difference is greater than either minuend or subtrahend
   b) If the difference is greater than the **minuend.**
   c) If the difference is greater than the subtrahend
   d) None of these

The Seven Great Ideas in Computer Architecture

→ Computer Architecture is the study of how computers are designed, focusing on performance, power efficiency, Cost, Reliability & Programmability

→ The seven great ideas are :

i) Abstraction to Simplify Design : Hide lower-level details to manage complexity

ii) Make the Common case fast : Optimize operations that occur most frequently

iii) Performance via Parallelism : Use multiple hardware resources for simultaneous tasking

iv) Performance via Pipelining : Overlap instruction execution steps to increase throughput

v) Performance via Prediction : Guess outcomes (like branches) to avoid stalls & improve performance

vi) Hierarchy of Memories : Organise memory into levels so frequently accessed data is in fast memory

vii) Dependability via Redundancy : Use extra hardware to detect & tolerate failures, increasing reliability

Now lets go in detail

→ Abstraction to Simplify Design

→ Abstraction means hiding lower-level details & providing simple interface

ex: High level languages → Hide assembly details

Instruction Set Architecture (ISA) → Hides hardware implementation

Operating System → Hide device-specific details

Virtual Memory → Hides Physical Memory Layout

→ Abstraction enables Reusability, Portability, Manageable complexity

ex: printf("Hello"); ⟶ You aren't concerned about registers, syscalls or hardware

→ Make the Common case fast

→ Not all operations occur equally often, so, we optimize frequent operations to get better performance

ex: Branch prediction, Caches, Simple arithmetic optimized heavily, Seperate instruction/data caches

→ This is also known as Amdahl's Law

→ Amdahl's Law tells us the speedup of a program from improving one part of it is limited by how much time that part originally took

$$\text{speedup} = \frac{1}{(1-f) + \frac{f}{s}}$$

where, f : fraction of execution time that can benefit from improvement

s : speedup of improved portion

Q. 1. What is the overall speedup if you make 10% of a program 90 times faster?

Q. 2. What is the overall speedup if you make 90% of a program 10 times faster

A.1 $\text{speedup} = \frac{1}{(1-0.1) + \frac{0.1}{90}} \simeq \frac{1}{0.9011} \simeq 1.11$

A.2 $\text{speedup} = \frac{1}{(1-0.9) + \frac{0.9}{90}} = \frac{1}{0.19} \simeq 5.26$

Fallacies : A mistaken belief - something that seems true but is actually false

Pitfall : A common error - something that can easily cause real mistakes in practice

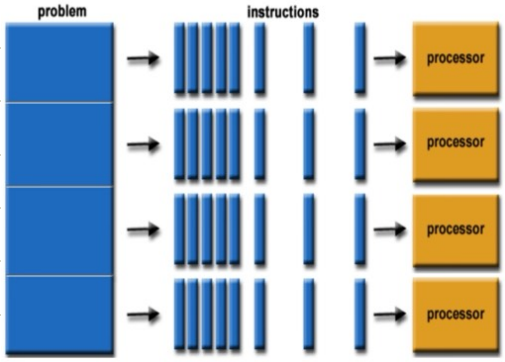
Corollary : A logical conclusion or guiding principle that follows from a main idea

← Click this for topic on them

→ Performance via Parallelism

→ It is to do more work in parallel

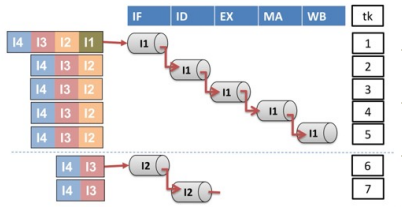
→ Program is broken into multiple smaller sequential computing operations some of which are performed simultaneously in parallel



ex: 5 stage pipeline in RISC-V , GPUs, Multicore CPUs

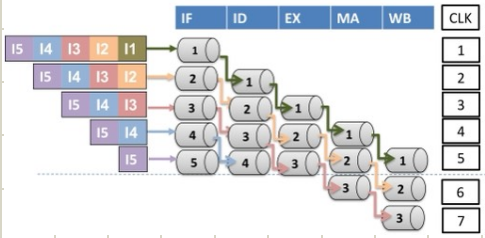
→ Performance via Pipelining

→ In a non-pipelined mode of execution, it must pass one instruction through all stages



Consumes lot of time

→ In a pipelined mode of execution, these stages are independantly performing the task in sequence



Consumes less time

→ Lets compare with an example

Instruction Fetch	T1=160n
Instruction Decoding	T2=120n
Execution	T3=130n
Memory Access	T4=100n
Write Back	T4=140n

For single cycle processor, $T_{CLK} = T_1 + T_2 + T_3 + T_4 + T_5$

$= 160 + 120 + 130 + 100 + 140 = 550ns$

For N instructions,

$T_{total} = N \times T_{CLK}$

For 5 stage pipeline processor, $T_{CLK} = \max(T_1, T_2, T_3, T_4, T_5)$

$= 160n$

For N instructions,

$T_{total} = (5 + N - 1) \times T_{CLK}$

→ Performance via Prediction

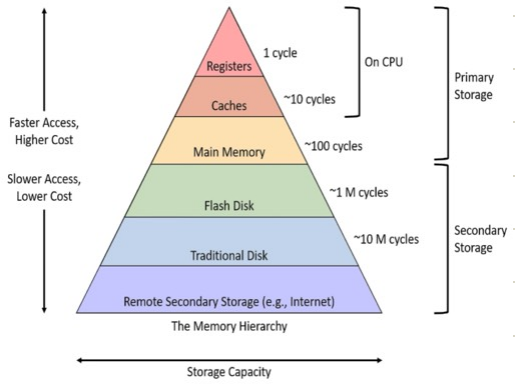
→ To avoid stalls, computer architecture use predictions

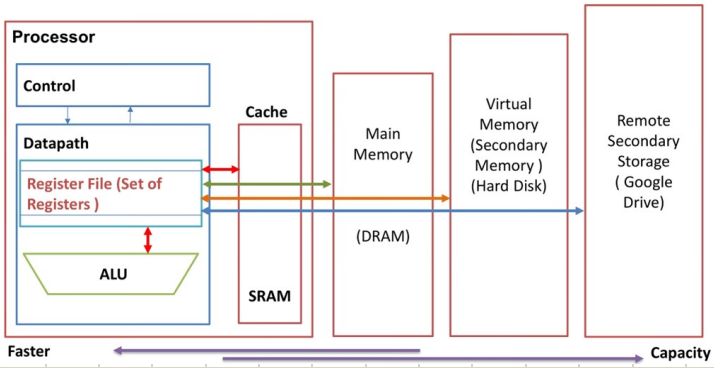
ex: Branch prediction, guess whether branch is taken to avoid stalling the pipeline

It can also be used for memory prefetching, speculative execution, cache replacement

→ Hierarchy of Memories

→ Memory hierarchy improves performance by using faster but smaller memory near the CPU





→ Keep the most frequent accessed data in the fastest memory

→ Memory should also be Fast, large & Cheap

→ Dependability via Redundancy

→ Use extra resources to tolerate faults making computers more dependable over fast

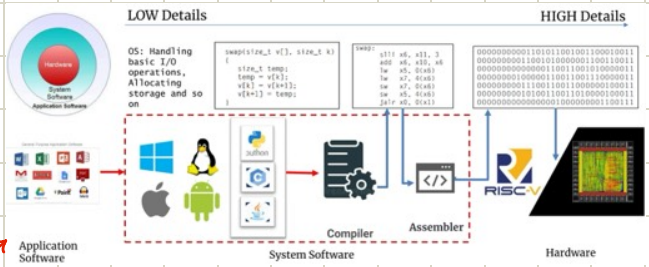
ex: ECC Memory (correct bit errors), RAID Storage (redundant disks), Dual power supplies, Triple Modular Redundancy

→ Redundancy helps:

→ Correct errors

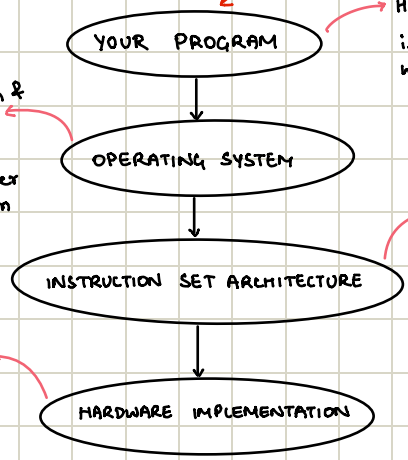
→ Detect failures

→ Continue operation even when some components fail



Below your Program

→ Lets understand the layers of abstraction



High level language which isn't understood by hardware (C, Python etc.,) written by user (YOU)

The OS provides protection & services like file handling, memory management etc., Its like a resource manager & intermediate b/w program and hardware

It is the boundary b/w software & hardware which defines: instruction formats, registers, data types, memory models etc.,

ex: RISC-V, ARM, x86, MIPS etc.,

A program compiled for an ISA will run on any hardware that implements that ISA

It is the physical implementation of ISA includes processor pipeline, ALU, control unit, memory controller etc., Microarchitecture can vary but still implement the same ISA

→ ISA provides stable abstraction so programs run correctly regardless of hardware

Technologies for building processors and memory

→ A transistor is an on/off switch controlled by electricity

→ The IC combines dozens to hundreds of transistors into single chip

→ This part explains evolution of semiconductor technology & how physical capability of building chips every year

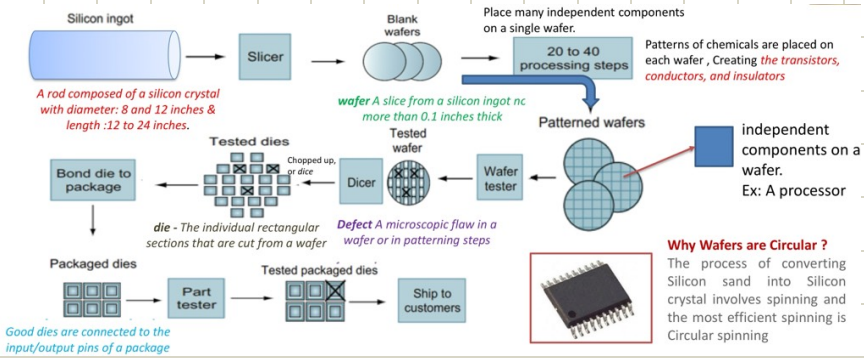
→ Moore's Law states that transistors per chip double approximately every 18-24 months

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
2020	Ultra large-scale integrated circuit	500,000,000,000

→ Relative performance per unit cost of technologies used in computers over time

→ **Chip Manufacturing Process**

→ Chips are manufactured on silicon wafers using multi-step photolithography process



→ Technology Node

Generation of manufacturing:

65nm → 45nm → 32nm → 14nm → 7nm → 5nm → 3nm

Smaller → Faster → Less Power → More Transistors

→ **Cost & Price of the Chip**

→ The cost of a chip comes from:

- Cost of wafer manufacturing
- Wafer Size
- Die Size
- Yield (Fraction of working dies)
- Packaging Cost

→ $\text{Dies per Wafer} = \frac{\text{Wafer area}}{\text{Die area}}$ (Wafer area {300 mm wafer} : $\pi(150)^2 = 70685\,mm^2$)

→ $\text{Yield} = \frac{1}{(1 + (\text{Defects per area} \times \text{Die area}))^N}$

(N : Process-complexity factor ⇒ For 130nm → N=4; For 28 nm → N = 7.5-9.5; For 16 nm → N = 10-14)

→ $\text{Cost per die} = \frac{\text{Cost per wafer}}{\text{Dies per wafer} \times \text{Yield}}$

→ Defects per Area derivation

$$\text{Yield}\left(1 + \text{defects per area} \times \frac{\text{die area}}{2}\right)^2 = 1$$

$$\sqrt{\text{Yield}}\left(1 + \text{defects per area} \times \frac{\text{die area}}{2}\right) = 1$$

$$\left(\sqrt{\text{Yield}} + \sqrt{\text{Yield}} \times \text{defects per area} \times \frac{\text{die area}}{2}\right) = 1$$

$$\sqrt{\text{Yield}} \times \text{defects per area} \times \frac{\text{die area}}{2} = 1 - \sqrt{\text{Yield}}$$

$$\text{defects per area} = \frac{1 - \sqrt{\text{Yield}}}{\sqrt{\text{Yield}} \times \frac{\text{die area}}{2}}$$

Q. **Question:** Find the number of dies per 300mm (30 cm) wafer for a die that is 1.5 cm on a side and for a die that is 1.0cm on a side.

A. Case (i): $\text{No. of dies} = \frac{\text{Wafer Area}}{\text{Die Area}} = \frac{\pi r^2}{1.5 \times 1.5} = \frac{3.14 \times 15^2}{1.5 \times 1.5} = 314$

Case (ii): $\text{No. of dies} = \frac{\text{Wafer Area}}{\text{Die Area}} = \frac{\pi r^2}{1.0 \times 1.0} = \frac{3.14 \times 15^2}{1.0 \times 1.0} = 706$

Q. **Question:** For wafer 300mm diameter, Find the die yield for dies that are 1.5 cm on a side and 1.0 cm on a side, assuming a defect density of 0.047 per cm^2 and N=12.

A. Case (i): Total Die Area $= (1.5 \times 1.5)\ cm^2 = 2.25\ cm^2$

$$\text{Yield} = \frac{1}{1 + (\text{defects per area} \times \text{die area})} = \left(\frac{1}{1 + (0.047 \times 2.25)^{12}}\right) = 0.2993$$

Out of 314, 94 dies will be good

Case (ii): Total Die Area $= (1.0 \times 1.0)\ cm^2 = 1.00\ cm^2$

$$\text{Yield} = \frac{1}{1 + (\text{defects per area} \times \text{die area})} = \left(\frac{1}{1 + (0.047 \times 1.00)^{12}}\right) = 0.5762$$

Out of 706, 407 dies will be good

Q. **Question:** Assume a 15 cm diameter wafer has a cost of 12, contains 84 dies, and has 0.020 defects/cm^2. Assume a 20 cm diameter wafer has a cost of 15, contains 100 dies, and has 0.031 defects/cm^2.

a. Find the yield for both wafers.
b. Find the cost per die for both wafers.
c. If the number of dies per wafer is increased by 10% and the defects per area unit increases by 15%, find the die area and yield.
d. Assume a fabrication process improves the yield from 0.92 to 0.95.Find the defects per area unit for each version of the technology given a die area of 200 mm^2.

A. a) Die area (15cm) $= \frac{\text{Wafer Area}}{\text{Dies per wafer}} = \frac{\pi \times 7.5^2}{84} = 2.1\ cm^2$

Yield (15cm) $= \frac{1}{\left(1 + (\text{Defect per Area} \times \frac{\text{Die Area}}{2})\right)^2} = \frac{1}{\left(1 + \left(0.020 \times \frac{2.1}{2}\right)\right)^2} = 0.9593$

Die area (20 cm) $= \frac{\text{Wafer Area}}{\text{Dies per wafer}} = \frac{\pi \times 10^2}{100} = 3.14\ cm^2$

Yield (20 cm) $= \frac{1}{\left(1 + (\text{Defect per Area} \times \frac{\text{Die Area}}{2})\right)^2} = \frac{1}{\left(1 + \left(0.031 \times \frac{3.14}{2}\right)\right)^2} = 0.9093$

b) Cost per die (15cm) $= \left(\frac{\text{Cost per wafer}}{\text{Dies per wafer} \times \text{Yield}}\right) = \left(\frac{12}{84 \times 0.9593}\right) = 0.1489$

Cost per die (20 cm) $= \left(\frac{\text{Cost per wafer}}{\text{Dies per wafer} \times \text{Yield}}\right) = \left(\frac{15}{100 \times 0.9093}\right) = 0.165$

c) Die area (15cm) $= \frac{\text{Wafer Area}}{\text{Dies per wafer}} = \frac{\pi \times 7.5^2}{84 \times 1.1} = 1.91\ cm^2$

Yield (15cm) $= \frac{1}{\left(1 + (\text{Defect per Area} \times \frac{\text{Die Area}}{2})\right)^2} = \frac{1}{\left(1 + \left(0.020 \times 1.15 \times \frac{1.91}{2}\right)\right)^2} = 0.957$

Die area (20 cm) $= \frac{\text{Wafer Area}}{\text{Dies per wafer}} = \frac{\pi \times 10^2}{100 \times 1.1} = 2.856\ cm^2$

Yield (20 cm) $= \frac{1}{\left(1 + (\text{Defect per Area} \times \frac{\text{Die Area}}{2})\right)^2} = \frac{1}{\left(1 + \left(0.031 \times 1.15 \times \frac{2.856}{2}\right)\right)^2} = 0.905$

d) defects per area (y=0.92) $= \frac{1 - \sqrt{\text{Yield}}}{\sqrt{\text{Yield}} \times \frac{\text{die area}}{2}} = \frac{1 - \sqrt{0.92}}{\sqrt{0.92} \times \frac{2}{2}} = 0.043$ defects/cm^2

defects per area (y=0.95) $= \frac{1 - \sqrt{\text{Yield}}}{\sqrt{\text{Yield}} \times \frac{\text{die area}}{2}} = \frac{1 - \sqrt{0.95}}{\sqrt{0.95} \times \frac{2}{2}} = 0.0259$ defects/cm^2

As yield ↑, defects per area ↓

Performance

→ It is how fast a computer runs a program

→ Execution time is the total time required for computer to complete a task

$$\text{Performance} = \frac{1}{\text{Execution Time}}$$

For 2 computers X and Y, If $\text{Performance}_x > \text{Performance}_y$

Then, $\text{Execution Time}_x < \text{Execution Time}_y$

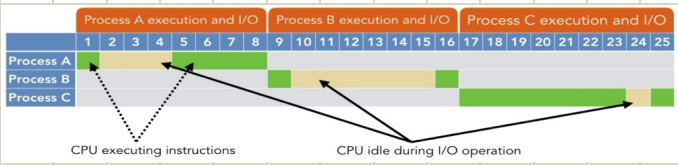
So, to relate performance of 2 different computers quantitatively,

$$\frac{\text{Performance}_x}{\text{Performance}_y} = n \Rightarrow \frac{\text{Execution Time}_y}{\text{Execution Time}_x} = n$$

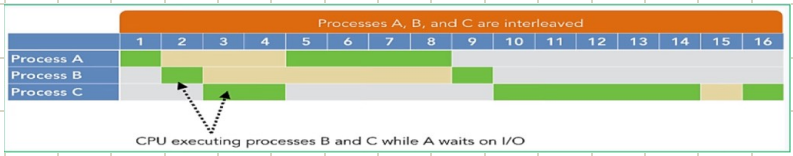
→ Throughput is the number of tasks completed per unit time

→ Wall Clock Time / Elapsed Time / Response Time is the total time required to complete a task including disk access, memory access, I/O activities, Operating System overhead - everything

→ Computers are often shared, however, a processor may work on several programs simultaneously In such cases, system tries to optimize throughput rather than minimizing elapsed time for 1 program



↑ Sequential uni-programming devices



↑ Multiprogramming Environment

■ : CPU Execution

■ : I/O Operations

→ Measuring Performance

→ CPU Execution Time is the time CPU spends on computing a particular task & doesn't include time spent waiting for I/O or running other programs.

It can be further divided into :

i) User CPU Time : Amount of time the processor spends in running application code

ii) System CPU Time : Amount of time the processor spends in running OS function related to code

Consider this as CPU Performance

$$\text{CPU execution time for a program} = \text{CPU clock cycles for a program} \times \text{Clock cycle time}$$

$$= \frac{\text{CPU clock cycles for a program}}{\text{Clock Rate}}$$

Q. **Computer A**

Our favorite program runs in 10 seconds on computer A, which has a 2 GHz clock.

Computer B

Computer designer build a computer, B, which will run this program in 6 seconds

The designer has determined that a substantial increase in the clock rate is possible, **but this increase will affect the rest of the CPU design, causing computer B to take 1.2 times as many clock cycles as computer A** for this program. What clock rate should we tell the designer to target?

A. For computer A, $\text{CPU time}_A = \dfrac{\text{CPU Clock cycles}_A}{\text{Clock Rate}_A}$

$$10 \text{ sec} = \frac{\text{CPU Clock cycles}_A}{2 \times 10^9 \text{ cycles/sec}} \Rightarrow \text{CPU Clock Cycles}_A = 2 \times 10^{10} \text{ Clock Cycles}$$

We are Given $\text{CPU Clock cycles}_B = 1.2 \times \text{CPU Clock cycles}_A$

For computer B, $\text{CPU time}_B = \dfrac{\text{CPU Clock cycles}_B}{\text{Clock Rate}_B} \Rightarrow \text{Clock Rate}_B = \dfrac{1.2 \times 2 \times 10^{10}}{6} = 4 \text{ GHz}$

$= 4 \times 10^9$ cycles/sec

→ Instruction Performance

→ Till now we haven't come to the discussion of no. of instructions.
Now lets express execution time in terms of no. of instructions.

→ We know that,

CPU execution time for a program = CPU clock cycles for a program × Clock cycle time

= (Instructions for a program × Average clock cycles per instruction) × Clock cycle time

→ Clock cycles per instruction

→ Average no. of clock cycles each instruction takes to execute

→ The reason we take average is because different instructions may take different amounts of time depending on what they do.

Q. Suppose we have two implementations of the same instruction set architecture.

Computer A

Computer A has a clock cycle time of 250ps and a CPI of 2.0 for some program

Computer B

computer B has a clock cycle time of 500 ps and a CPI of 1.2 for the same program

Which computer is faster for this program and by how much?

A. $\text{CPU clock cycles}_A = I \times 2.0$

$\text{CPU clock cycles}_B = I \times 1.2$

$\text{CPU Time}_A = \text{CPU clock cycles}_A \times \text{Clock cycle time} = I \times 2.0 \times 250\text{ps} = 500I \text{ ps}$

$\text{CPU Time}_B = \text{CPU clock cycles}_B \times \text{Clock cycle time} = I \times 1.2 \times 500\text{ps} = 600I \text{ ps}$

$$\frac{\text{CPU Performance}_A}{\text{CPU Performance}_B} = \frac{\text{Execution Time}_B}{\text{Execution Time}_A} = \frac{600I}{500I} = 1.2$$

Computer A is 1.2 times faster than computer B

→ **Classic CPU Performance Equation**

→ Basic performance in terms of instruction count, CPI & clock cycle time :

$$\text{CPU Time} = \text{Instruction Count} \times \text{CPI} \times \text{Clock cycle time}$$

$$= \frac{\text{Instruction Count} \times \text{CPI}}{\text{Clock Rate}}$$

Q. A compiler designer is trying to decide between two code sequences for a computer. The hardware designers have supplied the following facts:

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

For a particular high-level language statement, the compiler writer is considering two code sequences that require the following instruction counts:

Code sequence	Instruction counts for each instruction class		
	A	B	C
1	2	1	2
2	4	1	1

Which code sequence executes the most instructions? ① Which will be faster? ② What is the CPI for each sequence? ③

A. Code sequence 1 : No. of instructions = 2+1+2 = 5

Code sequence 2 : No. of instructions = 4+1+1 = 6 ⟶ Executes the most instructions ①

Total no. of clock cycles for sequence 1 = (2×1) + (1×2) + (2×3) = 10 clock cycles

Total no. of clock cycles for sequence 2 = (4×1) + (1×2) + (1×3) = 9 clock cycles ⟶ Faster ②

$$CPI_1 = \frac{\text{CPU clock cycles}_1}{\text{Instruction Count}_1} = \frac{10}{5} = 2.0$$

$$CPI_2 = \frac{\text{CPU clock cycles}_2}{\text{Instruction Count}_2} = \frac{9}{6} = 1.5$$

③

→ **Instructions per Clock Cycle**

→ Very straightforward

$$IPC = \frac{1}{CPI}$$

→ CISC : Min CPI = 1

Single Cycle Processor : CPI = 1

Pipelined Processor : CPI > 1

Some processors (like dual core) : CPI < 1

Summary of formulas

→ Total execution time E_{Total} = No. of CLK cycles to execute prog * T_{CLK} = No. of CLK cycles to execute prog / f_{CLK}

→ No. of CLK cycles to execute prog = Instruction Count * CPI

→ Total execution time = IC × CPI × T_{CLK} = IC × CPI / f_{CLK}

→ CPI = $E_{Total} \times f_{CLK} / IC$ = $E_{Total} / (IC \times T_{CLK})$

→ If diff class of Inst.s are given, $E_{Total} = \sum_i (IC_i \times CPI_i) \times T_{CLK}$

→ Inst.s per second = IPC × f_{CLK}

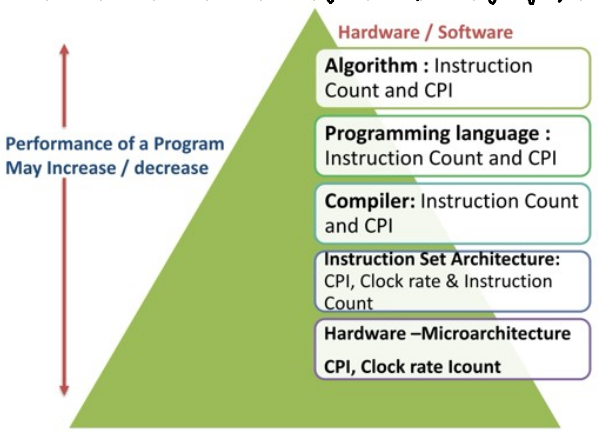
→ DONT NEGLECT, PRACTICE QUESTIONS FROM THE QB I HAVE UPLOADED!

→ Basic components of performance & how they're measured

Components of performance	Units of measure
CPU execution time for a program	Seconds for the program
Instruction count	Instructions executed for the program
Clock cycles per instruction (CPI)	Average number of clock cycles per instruction
Clock cycle time	Seconds per clock cycle

→ Parameters affecting Performance of a Program

→ Parameters are algorithm, language, compiler, architecture & actual hardware



→ Turbo Mode

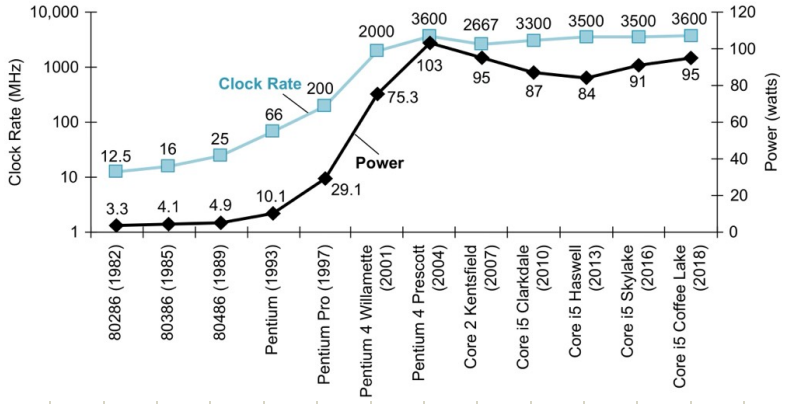
→ Even though clock cycle time has traditionally been fixed, to save energy or boost performance temporarily, processors can vary their clock rates

ex: Intel Core i7 temporarily increases clock rate by about 10% until chip gets warm They call this turbo mode

→ Hence, we need to use the average clock rate for a program

Processor Number	Cores/ Threads	Base Freq (GHz)	Intel Turbo Boost Technology 2.0		
			Max Single Core Turbo (GHz)	Max Dual Core Turbo (GHz)	Max Quad Core Turbo (GHz)
Intel® Core™ i7 Processors S-Processor Line)					
i7-6700	*4/8*	*3.4*	*4.0*	*3.9*	*3.7*

Power Wall



→ The above timeline shows clock rate & power for Intel x86 microprocessors
We can see at around 2004, both clock rate & power have stopped increasing
Because both are correlated & eventually designers hit a practical limit on cooling

→ The Pentium 4 Prescott suffered thermal issues & Intel abandoned high-frequency approach

→ We have to measure energy usage because of the devices we use are heavily reliant on energy usage

→ Modern chips use CMOS & dominant energy cost is dynamic energy (Energy consumed to switch 1 to 0 & vice-versa)
Dynamic energy depends on : Energy $\propto$ Capacitive Load $\propto$ Voltage^2
So, Power $\propto C \propto V^2 \propto f$
C : Capacitive Load
V : Voltage
f : clock frequency

Q. Suppose we developed a new, simpler processor that has 85% of the capacitive load of the more complex older processor. Further, assume that it can adjust voltage so that it can reduce voltage 15% compared to processor B, which results in a 15% shrink in frequency. What is the impact on dynamic power?

A. $$\frac{\text{Power}_{new}}{\text{Power}_{old}} = \frac{\text{Capacitive Load} \times 0.85 \times \text{Voltage} \times 0.85^2 \times \text{Freq} \times 0.85}{\text{Capacitive Load} \times \text{Voltage} \times \text{Freq}} = 0.85^4 = 0.52$$

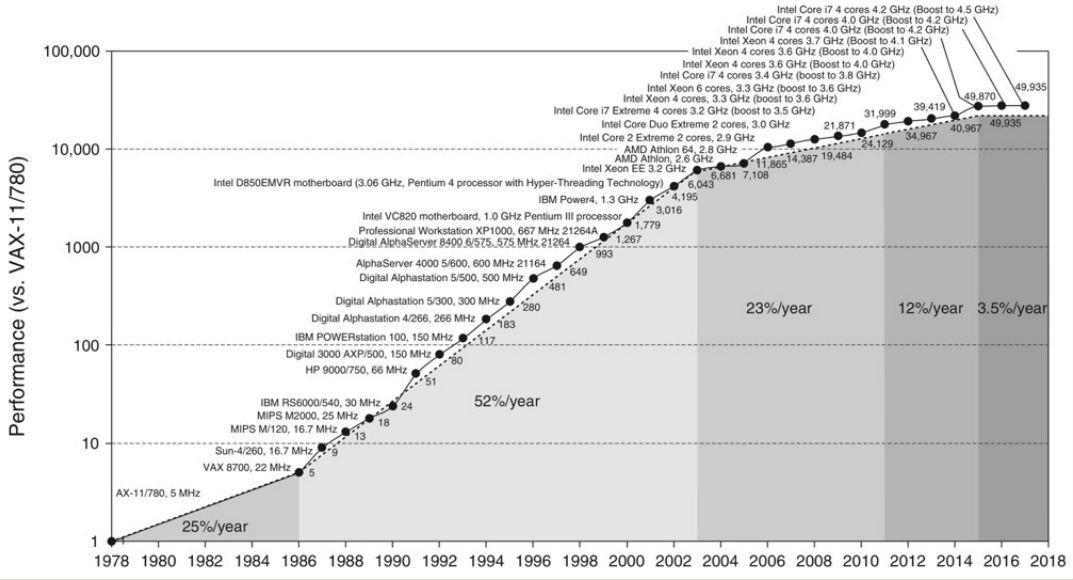
Hence, new processor uses around half the power of old processor

→ Modern chips suffer from leakage power, where transistors leak current even when OFF
& lowering voltage makes leakage worse, so voltage scaling has practically stopped

→ Designers use heavy cooling, but its not practical & expensive, so to turn off unused parts chips use clock gating & power gating

→ Power wall forced to abandon high-freq designs & shift to multicore processors, parallelism & energy efficient methods

Switch from Uniprocessors to Multiprocessors



→ For decades (mid 1980s to early 2000s), performance of processor increased rapidly due to higher clock rates, better microarchitectures, improved Instruction-level Parallelism
BUT after 2002, uniprocessor improvements slowed down because of Power wall, Limited ILP etc.,

→ Strategy had to be changed, and all major manufactures started shipping Multicore processors (One chip consisting multiple CPUs or "Cores")

→ Instead of making one chip faster, put several efficient cores in one chip

→ This gives better throughput, Better energy efficiency, Avoids power wall.

→ The main impact on programmers was that single core processors had only hardware improvements so not much change in source code.
But in case of multi-core, programs must be rewritten or parallelized

→ Work has to be split across threads, must handle race conditions & synchronization issues

→ The difficulties faced are:

i) Scheduling : Assignin
ii) Load Balancing
iii) Synchronization time
iv) Communication overhead
v) Dependencies

Benchmarking intel i7

→ A user usually measures performance by running own workload but most user can't so we use benchmarks

→ Benchmarks are selected programs that represent typical workloads
They help architects identify common case to optimize

→ SPEC or System Performative Evaluation Cooperative is the industry consortium
It creates standard benchmark suites to evaluate computer performance

→

Description	Name	Instruction Count x 10^9	CPI	Clock cycle time (seconds x 10^{-9})	Execution Time (seconds)	Reference Time (seconds)	SPECratio
Perl interpreter	perlbench	2684	0.42	0.556	627	1774	2.83
GNU C compiler	gcc	2322	0.67	0.556	863	3976	4.61
Route planning	mcf	1786	1.22	0.556	1215	4721	3.89
Discrete Event simulation - computer network	omnetpp	1107	0.82	0.556	507	1630	3.21
XML to HTML conversion via XSLT	xalancbmk	1314	0.75	0.556	549	1417	2.58
Video compression	x264	4488	0.32	0.556	813	1763	2.17
Artificial Intelligence: alpha-beta tree search (Chess)	deepsjeng	2216	0.57	0.556	698	1432	2.05
Artificial Intelligence: Monte Carlo tree search (Go)	leela	2236	0.79	0.556	987	1703	1.73
Artificial Intelligence: recursive solution generator (Sudoku)	exchange2	6683	0.46	0.556	1718	2939	1.71
General data compression	xz	8533	1.32	0.556	6290	6182	0.98
Geometric mean	–	–	–	–	–	–	2.36

⇒ SPECspeed 2017 integer benchmarks

First : SPEC89

Latest : SPEC CPU2017

10 integer benchmarks : SPEC speed 2017 integer

13 floating-point benchmarks : CFP2006

→ SPEC programs are large, complex & real applications designed to stress CPU + memory hierarchy

→ For each benchmark, SPEC records IC, CPI, Clock cycle time, Execution Time, SPEC ratio

→ SPEC ratio = $\frac{\text{Execution time of reference processor}}{\text{Execution time of evaluated computer}}$

↳ Bigger numeric results indicate faster machine
It normalizes execution time so all results are comparable
& each benchmark gets its own SPEC ratio

→ Summary result (SPEC2017 integer summary) = Geometric mean of all 10 SPEC ratios)
$= \left(\prod_{i=1}^{n} \text{Execution time ratio}\right)^{1/n}$

Note :
$\prod_{i=1}^{n} a_i = a_1 \times a_2 \times a_3 \times \dots \times a_n$

→ SPEC Power Benchmark

→ We know about the importance of Power,
So SPEC added a benchmark to measure power

→ It measures energy efficiency of servers
Uses Java business applications (SPECjbb-like workload)
Reports power & performance at diff. load levels & at each level, ssj-ops (Throughput) & power (Watts consumed)

ex :

Target Load %	Performance (ssj_ops)	Average Power (watts)
100%	4,864,136	347
90%	4,389,196	312
80%	3,905,724	278
70%	3,418,737	241
60%	2,925,811	212
50%	2,439,017	183
40%	1,951,394	160
30%	1,461,411	141
20%	974,045	128
10%	485,973	115
0%	0	48
Overall Sum	26,815,444	2,165
∑ssj_ops / ∑power =		12,385

At 10% load ⇒ Low throughput, Low power

At 100% load ⇒ High throughput, High power

Here, overall ssj-ops per watt = $\frac{\sum_{i=0}^{10} \text{ssj-ops}_i}{\sum_{i=0}^{10} \text{power}_i}$

Going Faster : Matrix Multiplication in Python

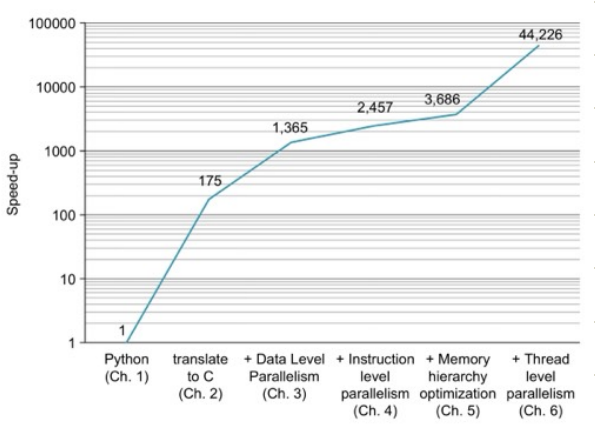
→ The python program to multiply 960 x 960 matrix

```
for i in xrange(n):
    for j in xrange(n):
        for k in xrange(n):
            C[i][j] += A[i][k] * B[k][j]
```

Run this on 96-core Google Cloud Server still takes 5 minutes

If this was 4096 x 4096, it would take 6 hours

This is because python is really slow



→ We can perform optimizations :

i) Rewriting program in C → 200x speedup

ii) Add SIMD instructions → 8x speedup

iii) Loop unrolling → 2x speedup

iv) Cache Blocking → 1.5x speedup

v) Multicore parallelism → 12-17x speedup

Together almost 50000x original python code

Some Fallacies & Pitfalls

(Click this for meaning → )

→ Pitfall 1: Expecting improvement in one component to translate to proportional overall improvement

→ Improving one part gives limited gains unless that part dominates total execution time

→ Amdahl's Law states,

$$\text{Execution time after improvement} = \frac{\text{Execution time of affected part}}{\text{Speedup}} + \text{Execution time of unaffected part}$$

ex: Program takes 100s, Multiply operation takes 80%, and we want 5x speedup

So new time should be 20s ⇒ $20 = \frac{80}{n} + 20$ which is impossible

→ Speedup is limited by fraction of time that improved part is used.

→ Fallacy 1: Computers at low utilization use low power

→ Modern servers use 33% peak power even at 10% utilization, & real datacenters workload b/w 10 to 50%

→ This is because of power leakage, Power overhead, Inefficient power supplies

→ Goal is to make use X% of peak power at X% workload. Leads to massive savings

→ Fallacy 2: Performance & Energy efficiency are unrelated goals

→ Energy = Power x Time

→ When program is run, entire system consumes power, so if it finished faster it runs for less time

→ Pitfall 2: Using incomplete performance metrics

→ CPU performance eqn ⇒ CPU Time = IC x CPI x Cycle Time

Common mistake is to use only 2 out of 3 factors to compare performance

→ MIPS is instruction execution rate ⇒ $\text{MIPS} = \frac{\text{Instruction Count}}{\text{Execution Time} \times 10^6}$

→ The problem with MIPS is instruction sets differ, MIPS varies by program on same machine, MIPS can increase if performance worsens

ex: Comp A, IC = 10B, clk rate = 4GHz, CPI = 1.0 & Comp B, IC = 8B, clk rate = 4GHz, CPI = 1.1. Even tho B has less instrs, CPI is worse so MIPS comparision is misleading