

# CS 135: Computer Architecture 1


**Instructor: Prof. Bhagi Narahari**  
*Dept. of Computer Science*  
 Course URL: [www.seas.gwu.edu/~narahari/cs135/](http://www.seas.gwu.edu/~narahari/cs135/)



## Course Objectives: Where are we....

---

CS 135: Computer Architecture, Bhagi Narahari




### An Important Idea: what are Computers meant to do ?

---

- Solve problems that are described in English (or Greek or French or Hindi or Chinese or ...) and using a box filled with electrons and magnetism to accomplish the task.

CS 135: Computer Architecture, Bhagi Narahari



### Problem Transformation - levels of abstraction


---

The desired behavior:  
the application

The building blocks:  
electronic devices

Natural Language
Algorithm
Program
Machine Architecture
Micro-architecture
Logic Circuits
Devices

CS 135: Computer Architecture, Bhagi Narahari




## Bits&Bytes to High level Programs

---

- User application written in high level language
- Program runs on a processor
  
- How are high level programs implemented on processor ?
  - Run-time stack, allocation of variables, translation of high level code to machine code
  - Map high level data structures to low level data structures
    - Struct to linear mapping in memory
- What else does software developer want after program is implemented correctly ?
  
- **PERFORMANCE!**

CS 135: Computer Architecture, Bhagi Narahari




## Performance of Programs

---

- “Complexity” of algorithms
- How good/efficient is your algorithm
  - Measure using Big-Oh notation:  $O(N \log N)$
- Next question : How well is the code executing on the machine ???????
  - Actual time to run the program
    - What are the factors that come into play
    - Where is the program and data stored
    - What are the actual machine instructions executed
- What are the technology trends and how do they play a role ?

CS 135: Computer Architecture, Bhagi Narahari




## Next Topics

---

- Performance of programs
  - What to measure
  - Model ?
  - Technology trends
- Memory organization basics
  - Memory hierarchy
  - Cache memory
  - Virtual memory

CS 135: Computer Architecture, Bhagi Narahari

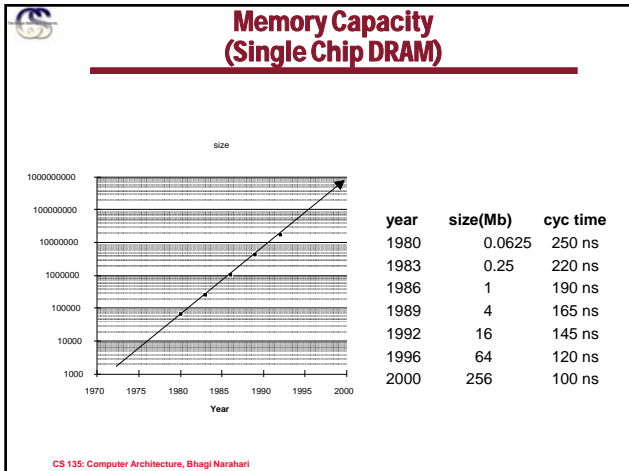
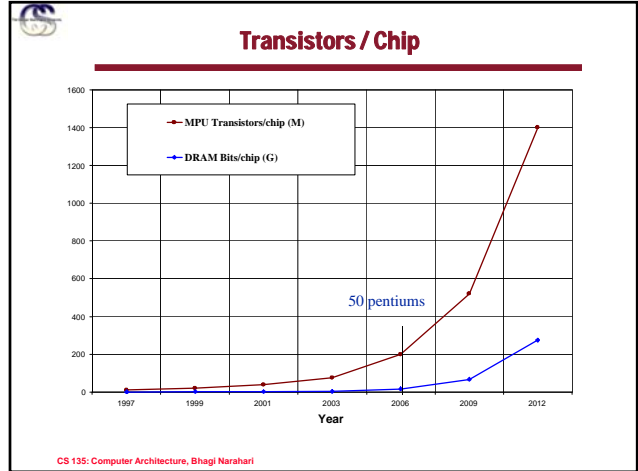
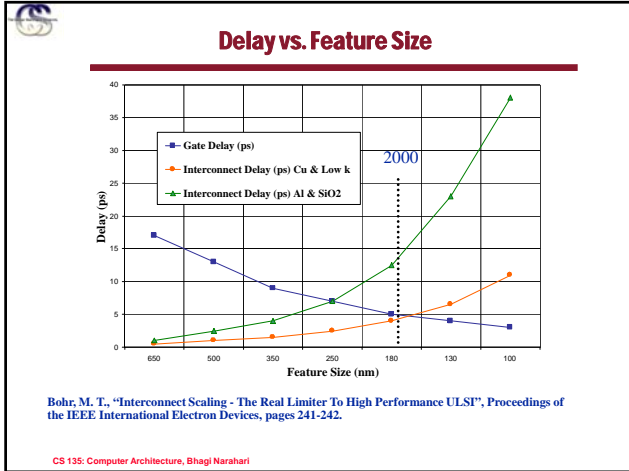


## Technology Trends

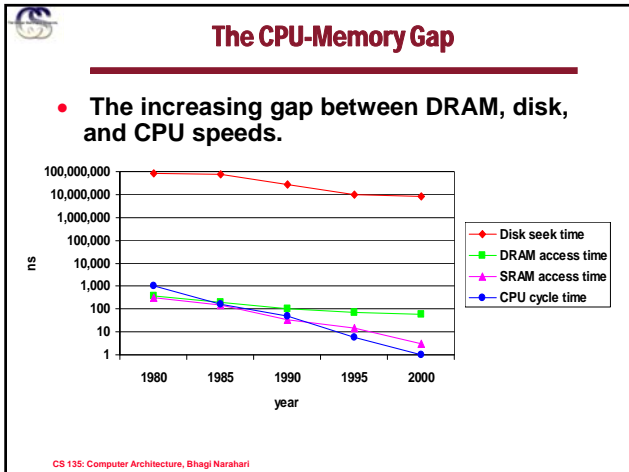
---

- Speed will depend on clock cycle (frequency) of the circuits
  - How fast can we switch the transistors
    - Feed the signal to the gate of MOS transistor, how long for the transistor to throw the switch
  - How large is the transistor – feature size
- **Moore’s Law**
  - Founder of Intel hypothesized on rate of increase in performance
    - It is not a law in the sense of laws of physics, etc.
  - Observations: performance doubles every 18 months
    - If you knew this, how would it guide your business decisions?
    - Case study: Apple Computers in '85

CS 135: Computer Architecture, Bhagi Narahari



- ### Performance Trends: Summary
- Workstation performance (measured in Spec Marks) improves roughly 50% per year (2X every 18 months)
    - Performance will include not just processor, but memory and disk I/O
  - Improvement in cost performance estimated at 70% per year
- CS 135: Computer Architecture, Bhagi Narahari



- ### Performance Metrics
- How do you measure performance?
    - Throughput
      - Number of tasks completed per time unit
    - Response time/Completion time of program
      - Time taken to complete a task/program
  - metric chosen depends on user community
    - System admin vs single user submitting homework
- CS 135: Computer Architecture, Bhagi Narahari

### The Bottom Line: Performance (and Cost)

Plane	DC to Paris	Speed	Passengers	Performance ?
Boeing 747	6.5 hours	610 mph	470	
BAD/Sud Concodre	3 hours	1350 mph	132	


CS 135: Computer Architecture, Bhagi Narahari

### The Bottom Line: Performance (and Cost)

Plane	DC to Paris	Speed	Passengers	Throughput (pmp)
Boeing 747	6.5 hours	610 mph	470	286,700
BAD/Sud Concodre	3 hours	1350 mph	132	178,200

- Time to run the task (Execution Time/Response Time/Latency)
  - Time to travel from DC to Paris
- Tasks per unit time (Throughput/Bandwidth)
  - Passenger miles per hour; how many passengers transported per unit time

CS 135: Computer Architecture, Bhagi Narahari




## How to Model Performance

---

- The asymptotic complexity – i.e., the “big O” notation !
  - Time =  $O(f(n))$  : function of the size of the input
  - Sorting  $O(n \log n)$
  - Naïve matrix multiplication:  $O(n^3)$
- This measures the efficiency of your algorithm
  - i.e., how well have you broken down your problem and solved it
    - Is this enough when we talk of actual time measured on the processor ???

CS 135: Computer Architecture, Bhagi Narahari




## Program Performance: The Great Reality – Our focus

---

- *There's more to performance than asymptotic complexity*
- Must optimize at multiple levels:
  - algorithm, data representations, procedures, and loops
- Must understand system to optimize performance
  - How programs are compiled and executed
  - How is data stored
  - What data structures are used
  - How to measure program performance and identify bottlenecks
  - How to improve performance without destroying code modularity and generality

CS 135: Computer Architecture, Bhagi Narahari




## Processor time: how to measure ?

---

- Number of clock cycles it takes to complete the execution of your program
- What is your program
  - A number of instructions
    - Different types: load, store, ALU, branch
  - Stored in memory
  - Executed on the CPU

CS 135: Computer Architecture, Bhagi Narahari



## Aspects of CPU Performance


---

<b>CPU time</b>	=	$\frac{\text{Seconds}}{\text{Program}}$	=	$\frac{\text{Instructions}}{\text{Program}} \times$	$\frac{\text{Cycles}}{\text{Instruction}} \times$	$\frac{\text{Seconds}}{\text{Cycle}}$
-----------------	---	-----------------------------------------	---	-----------------------------------------------------	---------------------------------------------------	---------------------------------------

**$CPU = IC * CPI * CIk$**

You will see more of this in CS 136 (?)

CS 135: Computer Architecture, Bhagi Narahari




## CPI

---

- **Cycles per instruction**
  - Different instructions may take different time
  - Example in LC 3 ?
- **We observed that not every instruction needs to go through all the instruction execution steps**
  - Eg: no need to calculate effective address, fetch from memory or registers
  - Reality: different times associated with different operations
    - Especially true of memory operations

CS 135: Computer Architecture, Bhagi Narahari




## Average CPI

---

- **Application has an “instruction mix”**
  - Profile of application instruction types
  - ALU, Load/Store (memory), Branch, Jumps, etc.
  - $x_1, x_2, x_3, \dots$  as percentage ( $x_1=0.4$ )
- **Processor has CPI for each type of instruction**
  - Example: ALU=1.0, Load/Store=2.0, etc.
  - $t_1, t_2, t_3, \dots$
- **What is effective CPI ?**
- **Weighted average**
  - $CPI = x_1 * t_1 + x_2 * t_2 + \dots$

CS 135: Computer Architecture, Bhagi Narahari



## CPI: Cycles per instruction

---


- **Depends on the instruction**

$CPI_i = \text{Execution time of instruction } i / \text{Cycle time}$
- **Average cycles per instruction**

$$CPI = \sum_{i=1}^n CPI_i * F_i \quad \text{where } F_i = \frac{IC_i}{IC_{tot}}$$
- **Example:**

Op	Freq	Cycles	$CPI_i$	%time
ALU	50%	1	0.5	33%
Load	20%	2	0.4	27%
Store	10%	2	0.2	13%
Branch	20%	2	0.4	27%
$CPI_{total}$			1.5	

CS 135: Computer Architecture, Bhagi Narahari



## Principles of Computer Architecture Design: Thumb Rules

---

- **Common case fast**
  - Focus on improving those instructions that are frequently used
  - Amdahl's Law
    - Fraction enhanced/optimized runs faster
- **Principle of Locality:**
  - program spends 90% of its time in 10% of code
    - Eg: word processing
  - Spatial: items near each other tend to be accessed
  - Temporal: recently used items tend to be used again
- **Concurrency**
  - Overlap the instruction execution steps
    - Pipeline processors – will see a LOT of this in CS 136
    - Multi-core processors

CS 135: Computer Architecture, Bhagi Narahari


## Amdahl's Law: Speedup

---

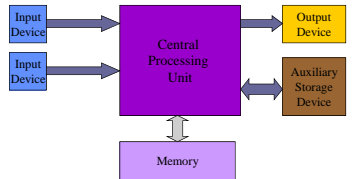
- Application takes X time
- How to run it faster
  - Enhance/optimize a portion of it
    - Which portion
    - Can we enhance all of it
    - Note that we are talking of solving the enhanced part in a different way, and possibly using different (more costly) resources
- Where to focus our optimizations ?
  - Look at return on investment
  - Code segments that take long time can give us the best returns
    - Profile your code to understand which parts are dominating

CS 135: Computer Architecture, Bhagi Narahari

## A Computer



---



```

graph LR
    ID1[Input Device] --> CPU[Central Processing Unit]
    ID2[Input Device] --> CPU
    CPU --> OD[Output Device]
    CPU <--> ASD[Auxiliary Storage Device]
    CPU <--> M[Memory]
  
```

The computer is composed of input devices, a central processing unit, a memory unit and output devices.

CS 135: Computer Architecture, Bhagi Narahari

## Recap

---

- CPU has two components:
  - Arithmetic and Logic Unit (ALU)
    - Performs arithmetic operations
    - Performs logical operations
  - Control Unit
    - Controls the action of the other computer components so that instructions are executed in the correct sequence
- Memory
  - Contains instructions and data
  - CPU requests data, data fetched from memory
    - How long does it take to fetch from memory ?

CS 135: Computer Architecture, Bhagi Narahari

## Memory Unit

---

- An ordered sequence of storage cells, each capable of holding a piece of data.
- Address space
  - Size of memory: N bit address space =  $2^N$  memory locations
- Addressability
  - Size of each memory location – k bits
  - Total size =  $k \cdot 2^N$  bits
- Assumption thus far: Processor/CPU gets data or instruction from some memory address (Inst fetch or Load/Store instruction)
  - But how is memory actually organized ?
  - Can everything we need fit into a memory that is close to the CPU ?

CS 135: Computer Architecture, Bhagi Narahari

**Where does run-time stack fit into this..**

- We looked at how variables in C are allocated memory addresses
  - Each function has activation record
  - Compiler takes care of allocation of memory addresses to variables
- Question now is: where are these memory addresses and how long does it take to fetch the contents into the processor register
  - LD R0, A
    - We know the address of A, but how long will it take to go into memory and fetch into register R0 ?

CS 135: Computer Architecture, Bhagi Narahari

**How is memory really organized ?**

- Many types of memory with different speeds
- Processor speed and memory speed mismatched
  - Data transferred between memory and processor
    - Instructions or data
- What does processor do while waiting for data to be transferred ?
  - Idle – processor is stalled leading to slowdown in speed and lower performance
- Why can't we have memory as fast as processor ?
  - Technology, cost, size
- What is the solution then ?

CS 135: Computer Architecture, Bhagi Narahari

**Memory Hierarchies**

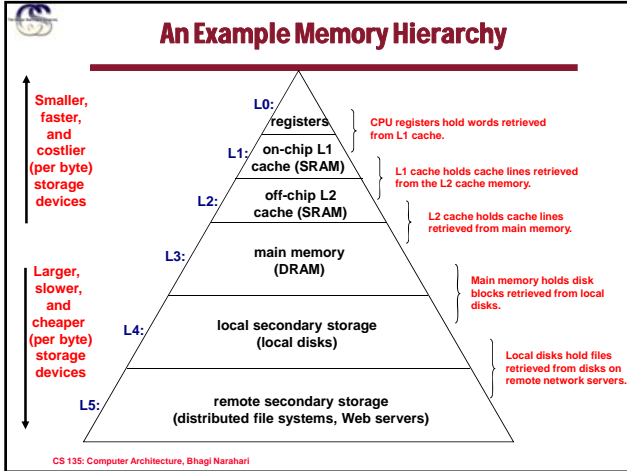
- Some fundamental and enduring properties of hardware and software:
  - Fast storage technologies cost more per byte and have less capacity.
  - The gap between CPU and main memory speed is widening.
  - Well-written programs tend to exhibit good locality.
- These fundamental properties complement each other beautifully.
- They suggest an approach for organizing memory and storage systems known as a **memory hierarchy**.

CS 135: Computer Architecture, Bhagi Narahari

The diagram shows a desk with a drawer and a cabinet. A 'Register file' is on the desk, with an arrow pointing to it labeled 'Access drawer in 5 s'. A 'Cache memory' is also on the desk, with an arrow pointing to it labeled 'Access desktop in 2 s'. A 'Main memory' is a cabinet, with an arrow pointing to it labeled 'Access cabinet in 30 s'.

**Items on a desktop (register) or in a drawer (cache) are more readily accessible than those in a file cabinet (main memory) or in a closet in another room**  
 keep more frequently accessed items on desktop, next frequent in drawer, etc. and things you need a lot less often in the closet in another room!

CS 135: Computer Architecture, Bhagi Narahari



### Memory Hierarchies

- **Key Principles**
  - **Locality** – most programs do not access code or data uniformly
  - **Smaller hardware is faster**
- **Goal**
  - Design a memory hierarchy “with cost almost as low as the cheapest level of the hierarchy and speed almost as fast as the fastest level”
    - This implies that we be clever about keeping more likely used data as “close” to the CPU as possible
- **Levels provide subsets**
  - Anything (data) found in a particular level is also found in the next level below.
  - Each level maps from a slower, larger memory to a smaller but faster memory

CS 135: Computer Architecture, Bhagi Narahari

### Locality

- **Principle of Locality:**
  - Programs tend to reuse data and instructions near those they have used recently, or that were recently referenced themselves.
  - **Temporal locality:** Recently referenced items are likely to be referenced in the near future.
  - **Spatial locality:** Items with nearby addresses tend to be referenced close together in time.

CS 135: Computer Architecture, Bhagi Narahari

### Locality

**Locality Example:**

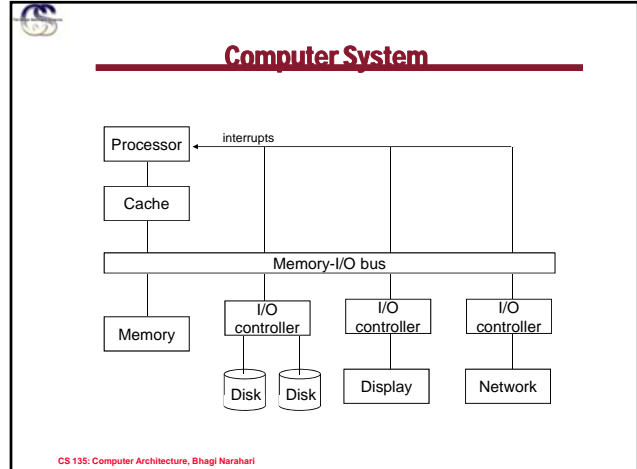
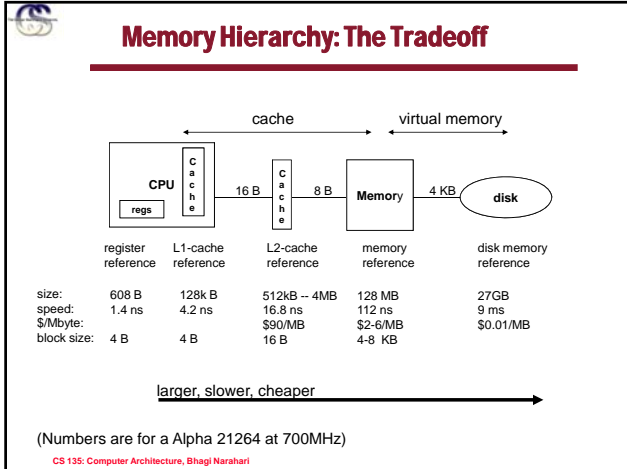
```

sum = 0;
for (i = 0; i < n; i++)
    sum += a[i];
return sum;

```


- **Data**
  - Reference array elements in succession (stride-1 reference pattern): **Spatial locality**
  - Reference `sum` each iteration: **Temporal locality**
- **Instructions**
  - Reference instructions in sequence: **Spatial locality**
  - Cycle through loop repeatedly: **Temporal locality**

CS 135: Computer Architecture, Bhagi Narahari



- ### Memory Hierarchy: Key concepts
- Details provided in notes on Memory design
    - What is a Cache memory
    - How does Memory access and Disk access work ?
    - How should we organize the memory hierarchy
  - How can we rewrite our code to improve performance
    - Based on memory access, type of instructions, etc.
- CS 135: Computer Architecture, Bhagi Narahari

- ### Simple Model of Memory – for now...
- Sequence of addresses
    - How many ?
  - CPU generates request for memory location – i.e., an address
    - How long does it take to get this data ?
      - Depends where it is in the Memory hierarchy
  - Simplified Model for memory hierarchy:
    - small amount of On-chip Cache memory
    - Larger amount of off-chip Main memory
    - Huge Disk
- CS 135: Computer Architecture, Bhagi Narahari




## Memory Access times

---

- **memory access time**
  - On-chip Cache takes 1 processor cycle
  - Main memory takes a number (10-50) processor cycles
  - Disk takes a huge amount
- **Simple model we will use for now:**
  - Memory = Cache + Main memory
  - Small size Cache = not everything fits in it
- **Simplified Cache organization:**
  - Cache consists of a set of blocks each of some number of bytes
  - Only a block can be fetched into and out of cache
  - Eg; if block is 16 bytes, then load 16 bytes into cache
    - Cannot load a single byte

CS 135: Computer Architecture, Bhagi Narahari




## Memory Access times using Simplified Model

---

- **If data is found in Cache then time =1**
  - Called a **cache hit**
- **Else time is Main memory access time**
  - **Cache miss**, means read from next level
- **Note: need a 'control unit' to determine if location is in cache or not**
  - **Cache controller**
- **Why does concept of caching work ?**
  - **Principle of Locality**
    - Programs access data nearby, or data/instructions that were used recently

CS 135: Computer Architecture, Bhagi Narahari




## Summary: Memory Access time optimization

---

- **If each access to memory leads to a **cache hit** then time to fetch from memory is one cycle**
  - **Program performance is good!**
- **If each access to memory leads to a **cache miss** then time to fetch from memory is much larger than 1 cycle**
  - **Program performance is bad!**
- **Design Goal:**  
*How to arrange data/instructions so that we have as few cache misses as possible.*

CS 135: Computer Architecture, Bhagi Narahari





## Next: Back to program performance

---


- **Try to minimize number of compute cycles (CPU/ALU cycles)**
  - **How ?**
    - Minimize number of ALU operations
- **Try to minimize number of cache misses**
  - **How ?**
    - Use principle of locality!

CS 135: Computer Architecture, Bhagi Narahari

# CS 135: Computer Architecture 1

Instructor: Prof. Bhagi Narahari  
 Dept. of Computer Science  
 Course URL: [www.seas.gwu.edu/~narahari/cs135/](http://www.seas.gwu.edu/~narahari/cs135/)




## Code optimization for performance

---

- A quick look at some techniques that can improve the performance of your code
- Rewrite code to minimize processor cycles
  - But do not mess up the correctness!
  - Reduce number of instructions executed
  - Reduce the “complexity” of instructions
    - In real processors, different arithmetic operations can take different times
- Locality
  - Will improve memory performance
- In reality: Compiler does a lot of code optimizations...

CS 135: Computer Architecture, Bhagi Narahari




## Recall -- Processor time: how to measure ?

---

- Number of clock cycles it takes to complete the execution of your program
- What is your program
  - A number of instructions
    - Different types: load, store, ALU, branch
  - Stored in memory
  - Executed on the CPU

CS 135: Computer Architecture, Bhagi Narahari



## Aspects of CPU Performance

---

$\text{CPU time} = \frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Cycle}}$
---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

$$CPU = IC * CPI * CIK$$

CS 135: Computer Architecture, Bhagi Narahari

## CPI

---

- **Cycles per instruction**
  - Different instructions may take different time
  - In LC3, we observed that not every instruction needs to go through all the instruction execution steps
    - Eg: no need to calculate effective address, fetch from memory or registers
- **Reality: different times associated with different operations**
  - Especially true of memory operations

CS 135: Computer Architecture, Bhagi Narahari

## Average CPI

---

- **Application has an “instruction mix”**
  - Profile of application instruction types
  - ALU, Load/Store (memory), Branch, Jumps, etc.
  - $x_1, x_2, x_3, \dots$  as percentage ( $x_1=0.4$ )
- **Processor has CPI for each type of instruction**
  - Example: ALU=1.0, Load/Store=2.0, etc.
  - $t_1, t_2, t_3, \dots$
- **What is effective CPI ?**
- **Weighted average**
  - $CPI = x_1 * t_1 + x_2 * t_2 + \dots$

CS 135: Computer Architecture, Bhagi Narahari

## CPI: Cycles per instruction

---

- **Depends on the instruction**

$CPI_i = \text{Execution time of instruction } i / \text{Cycle time}$
- **Average cycles per instruction**

$$CPI = \sum_{i=1}^n CPI_i * F_i \quad \text{where } F_i = \frac{IC_i}{IC_{tot}}$$
- **Example:**

Op	Freq	Cycles	$CPI_i$	%time
ALU	50%	1	0.5	33%
Load	20%	2	0.4	27%
Store	10%	2	0.2	13%
Branch	20%	2	0.4	27%
CPI <sub>total</sub>			1.5	

CS 135: Computer Architecture, Bhagi Narahari


## Summary: Memory Access time optimization

---

- If each access to memory leads to a **cache hit** then time to fetch from memory is one cycle
  - **Program performance is good!**
- If each access to memory leads to a **cache miss** then time to fetch from memory is much larger than 1 cycle
  - **Program performance is bad!**
- **Design Goal:**

*How to arrange data/instructions so that we have as few cache misses as possible.*

CS 135: Computer Architecture, Bhagi Narahari




## program performance

---

- Try to minimize number of compute cycles (CPU/ALU cycles)
  - How ?
    - Minimize number of ALU operations
  
- Try to minimize number of cache misses
  - How ?
    - Use principle of locality!

CS 135: Computer Architecture, Bhagi Narahari




## Project 4 Information & Logistics

---

- **Topic: Performance Optimization**
  - Given code for Image Smoothing and Image Rotation, rewrite the code to make it run faster.
    - Use only techniques covered in class.
  
- **You must work alone**
  - Can discuss Group assignment 7 in team
    - Feeds into Project 4
  
- **You MUST test on Hobbes**
  - I will use this to test your solutions

CS 135: Computer Architecture, Bhagi Narahari




## Next..

---

- **CODE OPTIMIZATION**

CS 135: Computer Architecture, Bhagi Narahari



## Memory Organization: Summary

---

- Memory is a sequence of addresses
- Memory consists of
  - Cache
  - Main Memory
- Cache contains some of the memory
- Access to cache is fast
- Access to main memory is slow
- If you fetch from Cache, then time to fetch is small
- If item is found in cache then Cache Hit else Cache miss
  - Miss means go to main memory...more time

CS 135: Computer Architecture, Bhagi Narahari

## Code optimization for performance

---

- A quick look at some techniques that can improve the performance of your code
- Rewrite code to minimize processor cycles
  - But do not mess up the correctness!
  - Reduce number of instructions executed
  - Reduce the “complexity” of instructions
    - In real processors, different arithmetic operations can take different times
- Locality
  - Will improve memory performance

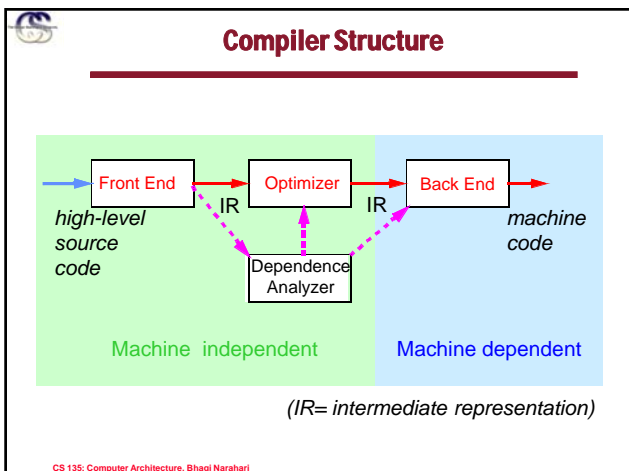
CS 135: Computer Architecture, Bhagi Narahari

## Compiler Tasks

---

- Code Translation
  - Source language → target language
    - FORTRAN → C
    - C → MIPS, PowerPC or Alpha machine code
    - MIPS binary → Alpha binary
- Code Optimization
  - Code runs faster
  - Match dynamic code behavior to static machine structure

CS 135: Computer Architecture, Bhagi Narahari




## Front End

---

- Lexical Analysis
  - Misspelling an identifier, keyword, or operator  
e.g. lex
- Syntax Analysis
  - Grammar errors, such as mismatched parentheses  
e.g. yacc
- Semantic Analysis
  - Type checking

CS 135: Computer Architecture, Bhagi Narahari



## Formal Model for Code Optimization ?

---

- Is it a hack job or is there a formal model underlying the various transformations that can help with designing a tool to optimize code ?
  - Need to make sure that transformed code is correct and does not change semantics of the original program.
  
- Graph theory: model program as a graph (Program dependence graph)
  - Model data and control dependencies
  - Any transformation should give us an isomorphic graph
    - Recall concept of Isomorphism/Homomorphism from CS124!!!

CS 135: Computer Architecture, Bhagi Narahari




## Machine-Independent Optimizations

---

- Optimizations you should do regardless of processor / compiler

CS 135: Computer Architecture, Bhagi Narahari




## Machine-Independent Optimizations

---

- Dataflow Analysis and Optimizations
  - Constant propagation
  - Copy propagation
  - Value numbering
  
- Elimination of common subexpression
- Dead code elimination
- Code motion
- Strength reduction
- Function/Procedure inlining

CS 135: Computer Architecture, Bhagi Narahari



## Code-Optimizing Transformations

---

- Constant folding
 

$(1 + 2)$	$\Rightarrow$	3
$(100 > 0)$	$\Rightarrow$	true
  
- Copy propagation
 

$x = b + c$		$x = b + c$
$z = y * x$	$\Rightarrow$	$z = y * (b + c)$
  
- Common subexpression
 

$x = b * c + 4$		$t = b * c$
$z = b * c - 1$	$\Rightarrow$	$x = t + 4$
		$z = t - 1$
  
- Dead code elimination
 

$x = 1$		
$x = b + c$		<i>or if x is not referred to at all</i>

CS 135: Computer Architecture, Bhagi Narahari

### Code Optimization Example

```

x = 1
y = a*b+3
z = a*b+x+z+2
x = 3
  
```

→ propagation

```

x = 1
y = a*b+3
z = a*b+1+z+2
x = 3
  
```

↓ constant folding

```

x = 1
y = a*b+3
z = a*b+3+z
x = 3
  
```

← dead code elimination

```

y = a*b+3
z = a*b+3+z
x = 3
  
```

↓ common subexpression

```

t = a*b+3
y = t
z = t+z
x = 3
  
```

CS 135: Computer Architecture, Bhagi Narahari

### Code Motion

- **Code Motion**
  - Reduce frequency with which computation performed
    - If it will always produce same result
    - Especially moving code out of loop
- **Move code between blocks**
  - eg. move loop invariant computations outside of loops
- **What does this reduce ?**

```

while (i < 100) {
    *p = x/y + i
    i = i + 1
}
  
```

→

```

t = x/y
while (i < 100) {
    *p = t + i
    i = i + 1
}
  
```

CS 135: Computer Architecture, Bhagi Narahari

### Code Motion

```

for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    a[n*i + j] = b[j];
  
```

→

```

for (i = 0; i < n; i++) {
    int ni = n*i;
    for (j = 0; j < n; j++)
        a[ni + j] = b[j];
}
  
```

CS 135: Computer Architecture, Bhagi Narahari

### Compiler-Generated Code Motion

- Most compilers do a good job with array code + simple loop structures
- **Code Generated by GCC**

```

for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    a[n*i + j] = b[j];
  
```

```

for (i = 0; i < n; i++)
{
    int ni = n*i;
    int *p = a+ni;
    for (j = 0; j < n;
j++)
        *p++ = b[j];
}
  
```

CS 135: Computer Architecture, Bhagi Narahari

## Reduction in Strength

- > Replace costly operation with simpler one
- > Shift, add instead of multiply or divide
  - 16\*x --> x << 4
  - > Utility is machine dependent
  - > Depends on cost of multiply or divide instruction
  - > On Pentium II or III, integer multiply only requires 4 CPU cycles
- > Recognize sequence of products

```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    a[n*i + j] = b[j];
```

→

```
int ni = 0;
for (i = 0; i < n; i++) {
  for (j = 0; j < n; j++)
    a[ni + j] = b[j];
  ni += n;
}
```

CS 135: Computer Architecture, Bhagi Narahari

## Strength Reduction

- Replace complex (and costly) expressions with simpler ones
  - > What does this reduce ?
- E.g.
  - a := b\*17 → a := (b<<4) + b
- E.g.
 

```
while (i < 100) {
  a[i] = i * 100;
  i = i + 1;
}
```

→

```
p = &a[i]
t = i * 100
while (i < 100) {
  *p = t
  t = t + 100
  p = p + 4
  i = i + 1
}
```

*loop invariant: &a[i]==p, i\*100==t*

CS 135: Computer Architecture, Bhagi Narahari

## Function Inlining

- What happens on a function call ?
  - > How are function calls implemented on the machine ?
  - > Is function call = one subroutine call ?
- Function call in C = number of instructions in machine code
  - > Create activation records, allocate memory
  - > Manipulate stack and frame pointers
- What happens if we replace function call with body of function ?
  - > Inline the function

CS 135: Computer Architecture, Bhagi Narahari

## Function Inlining

```
... int myfunc(int m,n)
x= myfunc(i,j) {
... return(m+n);
```

After inlining:

```
...
x = m+n
.....
```

CS 135: Computer Architecture, Bhagi Narahari

### Link with Memory organization... ..

- Let's use array data structures to guide our discussions
- Recall: accesses to cache better than accesses to main memory/disk
- Recall: Multidimensional Arrays

CS 135: Computer Architecture, Bhagi Narahari

### Declaration

```
int ia[3][4];
```

Number of Columns

Number of Rows

Address

Type

**Declaration at compile time  
i.e. size must be known**

CS 135: Computer Architecture, Bhagi Narahari

### How does a two dimensional array work?

	0	1	2	3
0				
1				
2				

How would you store it?

CS 135: Computer Architecture, Bhagi Narahari

### How would you store it?

	0	1	2	3
0				
1				
2				

**Column Major Order**

0,0	1,0	2,0	0,1	1,1	2,1	0,2	1,2	2,2	0,3	1,3	2,3
-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----

Column 0      Column 1      Column 2      Column 3

**Row Major Order**

0,0	0,1	0,2	0,3	1,0	1,1	1,2	1,3	2,0	2,1	2,2	2,3
-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----

Row 0      Row 1      Row 2

CS 135: Computer Architecture, Bhagi Narahari

## Advantage

- Using Row Major Order allows visualization as an array of arrays

```

ia[1]
0,0 0,1 0,2 0,3 1,0 1,1 1,2 1,3 2,0 2,1 2,2 2,3

ia[1][2]
0,0 0,1 0,2 0,3 1,0 1,1 1,2 1,3 2,0 2,1 2,2 2,3

```

CS 135: Computer Architecture, Bhagi Narahari

## Relating storage rules to performance

- C stores arrays in row-major format

CS 135: Computer Architecture, Bhagi Narahari

## Recall

- One Dimensional Array
 

```
int ia[6];
```
- Address of beginning of array:
 

```
ia ≡ &ia[0]
```
- Two Dimensional Array
 

```
int ia[3][6];
```
- Address of beginning of array:
 

```
ia = &ia[0][0]
```
- also
  - Address of row 0:
 

```
ia[0] = &ia[0][0]
```
  - Address of row 1:
 

```
ia[1] = &ia[1][0]
```
  - Address of row 2:
 

```
ia[2] = &ia[2][0]
```

CS 135: Computer Architecture, Bhagi Narahari

## Element Access

- Given a row and a column index
- How to calculate location?
- To skip over required number of rows:
 

```
row_index * sizeof(row)
row_index * Number_of_columns * sizeof(arr_type)
```
- This plus *address of array* gives address of first element of desired row
- Add `column_index * sizeof(arr_type)` to get actual desired element

```

0,0 0,1 0,2 0,3 1,0 1,1 1,2 1,3 2,0 2,1 2,2 2,3

```

CS 135: Computer Architecture, Bhagi Narahari

## Element Access

---

```

Element_Address =
    Array_Address +
    Row_Index * Num_Columns * Sizeof(Arr_Type) +
    Column_Index * Sizeof(Arr_Type)

Element_Address =
    Array_Address +
    (Row_Index * Num_Columns + Column_Index) *
    Sizeof(Arr_Type)

```

CS 135: Computer Architecture, Bhagi Narahari

## What's Up?

---

- Consider a one dimensional array
- If asked to determine the address of a given element does one need to know the size of the array?
- Consider a 2D array
- What is needed to calculate the address of a given element (i,j)?
  - offset =  $i * \text{\#columns} + j$

CS 135: Computer Architecture, Bhagi Narahari

## Locality of Access

---

- How are elements in the array accessed in your program ?

CS 135: Computer Architecture, Bhagi Narahari

## Locality

---

- Principle of Locality:
  - Programs tend to reuse data and instructions near those they have used recently, or that were recently referenced themselves.
  - **Temporal locality**: Recently referenced items are likely to be referenced in the near future.
  - **Spatial locality**: Items with nearby addresses tend to be referenced close together in time.
- Locality Example:
 


```

sum = 0;
for (i = 0; i < n; i++)
    sum += a[i];
return sum;

```

  - Data
    - Reference array elements in succession (stride-1 reference pattern): **Spatial locality**
    - Reference sum each iteration: **Temporal locality**
  - Instructions
    - Reference instructions in sequence: **Spatial locality**
    - Cycle through loop repeatedly: **Temporal locality**

CS 135: Computer Architecture, Bhagi Narahari




## Locality and performance

---

- **Recall: Memory = Cache + Main memory**
  - Cache contains small number of bytes
- **Recall: cache is arranged as a set of blocks**
  - Can only fetch block at a time
- **Example:**
  - Assume each cache block has 4 words
  - If you fetch a block with addresses {0,1,2,3}
  - If four successive instructions use locations 0,1,2,3 then we only have one cache miss (first time to fetch block into cache)
  - If four successive instructions use locations 0,4,8,12 then each time we have to fetch a new cache block
    - Each memory access is a access to main memory
- **Goal: have locality in memory accesses in the cache**

CS 135: Computer Architecture, Bhagi Narahari




## Locality

---

- **Claim:** Being able to look at code and get a qualitative sense of its locality is a key skill for a professional programmer.

CS 135: Computer Architecture, Bhagi Narahari



## Locality Example

---

- **Question:** Does this function have good locality?


```

int sumarraycols(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum
}

```

CS 135: Computer Architecture, Bhagi Narahari



## Locality Example

---

- **Question:** Does this function have good locality?


```

int sumarrayrows(int a[M][N])
{
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum
}

```


CS 135: Computer Architecture, Bhagi Narahari



## Improving Memory Access Times (Cache Performance) by Compiler Optimizations

- McFarling [1989] improve perf. By rewriting the software
- Instructions
  - Reorder procedures in memory so as to reduce cache misses
  - Code Profiling to look at cache misses(using tools they developed)
- Data
  - **Merging Arrays:** improve spatial locality by single array of compound elements vs. 2 arrays
  - **Loop Interchange:** change nesting of loops to access data in order stored in memory
  - **Loop Fusion:** Combine 2 independent loops that have same looping and some variables overlap
  - **Blocking:** Improve temporal locality by accessing “blocks” of data repeatedly vs. going down whole columns or rows

CS 135: Computer Architecture, Bhagi Narahari




## Compiler optimizations – merging arrays

- This works by improving spatial locality
- For example, some programs may reference multiple arrays of the same size at the same time
  - Could be bad – not enough locality
    - Accesses may interfere with one another in the cache – conflict misses
- A solution: **Generate a single, compound array...**

```

/* Before:*/      /* After */
int tag[SIZE]     struct merge {
int byte1[SIZE]  int tag;
int byte2[SIZE]  int byte1;
int dirty[size]  int byte2;
                 int dirty;
                 }
                 struct merge cache_block_entry[SIZE]
```

CS 135: Computer Architecture, Bhagi Narahari




## Merging Arrays Example

```

/* Before: 2 sequential arrays */
int val[SIZE];
int key[SIZE];
```

CS 135: Computer Architecture, Bhagi Narahari




## Merging Arrays Example

```

/* After: 1 array of stuctures */
struct merge {
  int val;
  int key;
};
struct merge merged_array[SIZE];
```

**Reducing conflicts between val & key;  
improve spatial locality**


CS 135: Computer Architecture, Bhagi Narahari



## Compiler optimizations – loop interchange

- Some programs have nested loops that access memory in non-sequential order
  - > Simply changing the order of the loops may make them access the data *in* sequential order...
- What's an example of this?
  - > Recall: C stores 2-D arrays in row-major format

CS 135: Computer Architecture, Bhagi Narahari




## Loop Interchange Example

```

/* Before */
for (k = 0; k < 100; k = k+1)
  for (j = 0; j < 100; j = j+1)
    for (i = 0; i < 5000; i = i+1)
      x[i][j] = 2 * x[i][j];
  
```

CS 135: Computer Architecture, Bhagi Narahari




## Loop Interchange Example

```

/* After */
for (k = 0; k < 100; k = k+1)
  for (i = 0; i < 5000; i = i+1)
    for (j = 0; j < 100; j = j+1)
      x[i][j] = 2 * x[i][j];
  
```

Sequential accesses instead of striding through memory every 100 words; improved spatial locality

CS 135: Computer Architecture, Bhagi Narahari



## Compiler optimizations – loop fusion

- This one's pretty obvious once you hear what it is...
- Seeks to take advantage of:
  - > Programs that have separate sections of code that access the same arrays in different loops
    - > Especially when the loops use common data
  - > The idea is to "fuse" the loops into one common loop
- What's the target of this optimization?

CS 135: Computer Architecture, Bhagi Narahari

### Loop Fusion Example

---

```

/* Before */
for (i = 0; i < N; i = i+1)
  for (j = 0; j < N; j = j+1)
    a[i][j] = 1/b[i][j] * c[i][j];
for (i = 0; i < N; i = i+1)
  for (j = 0; j < N; j = j+1)
    d[i][j] = a[i][j] + c[i][j];

```

CS 135: Computer Architecture, Bhagi Narahari

### Loop Fusion Example

---

```

/* After */
for (i = 0; i < N; i = i+1)
  for (j = 0; j < N; j = j+1)
  {
    a[i][j] = 1/b[i][j] * c[i][j];
    d[i][j] = a[i][j] + c[i][j];
  }

```

**2 misses per access to a & c vs. one miss per access; improve spatial locality**

CS 135: Computer Architecture, Bhagi Narahari

### Compiler Optimization: Blocking.

---

- Can you keep locality in all memory operations
- This is probably the most “famous” of compiler optimizations to improve cache performance
- Another common concept: **blocking**
  - Rewrite code to process blocks of data at a time
  - Size of block = ??? Size of cache block!!

The diagram shows a large yellow rectangle representing a 2D array. In the top-left corner, there are two smaller colored rectangles: a blue one on the left and a green one on the right, representing data blocks.


CS 135: Computer Architecture, Bhagi Narahari

### Compiler optimizations – blocking

---

- Tries to reduce misses by improving temporal locality and spatial locality
- To get a handle on this, you have to work through code on your own
- this is used mainly with arrays!
- Simplest case??
  - Row-major access

CS 135: Computer Architecture, Bhagi Narahari



## Blocking Example

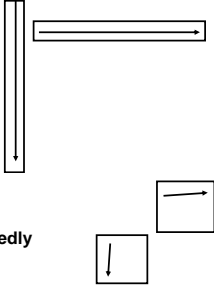
---

```


/* Before */
for (i = 0; i < N; i = i+1)
  for (j = 0; j < N; j = j+1)
    {r = 0;
     for (k = 0; k < N; k = k+1){
       r = r + y[i][k]*z[k][j];
     }
    x[i][j] = r;
  }

```

- **Two Inner Loops:**
  - > Read all NxN elements of z[]
  - > Read N elements of 1 row of y[] repeatedly
  - > Write N elements of 1 row of x[]
  
- **Idea: compute on BxB submatrix that fits**



CS 135: Computer Architecture, Bhagi Narahari



## Next...

---

- **Back to Memory Design**
  - > Focus on Cache design
  - > How does Cache memory work ?
  - > How are addresses mapped to Cache
  - > How to rewrite code to get better cache performance ?

CS 135: Computer Architecture, Bhagi Narahari