CS 33

Architecture and Optimization (1)
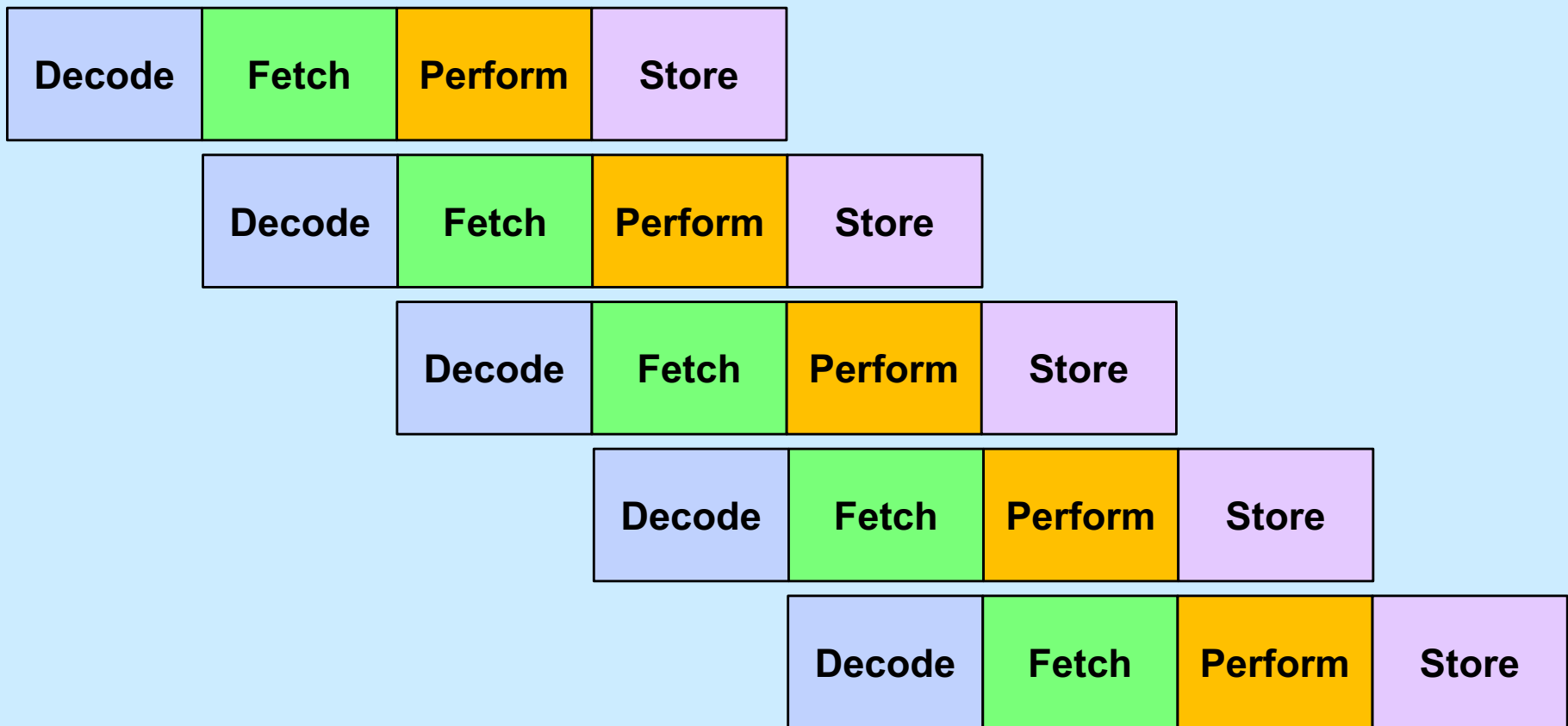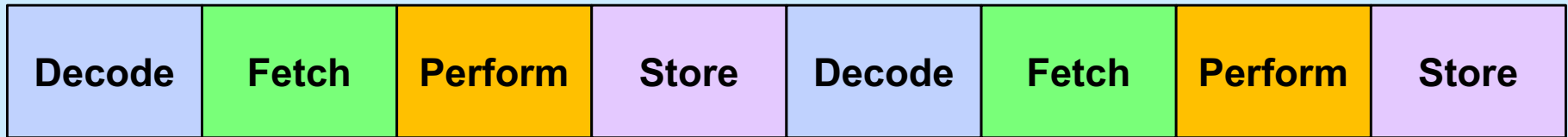
Simplistic View of Processor

```
while (true) {
    instruction = mem[rip];
    execute(instruction);
}
```

Some Details ...

```
void execute(instruction_t instruction) {
    decode(instruction, &opcode, &operands);
    fetch(operands, &in_operands);
    perform(opcode, in_operands, &out_operands);
    store(out_operands);
}
```
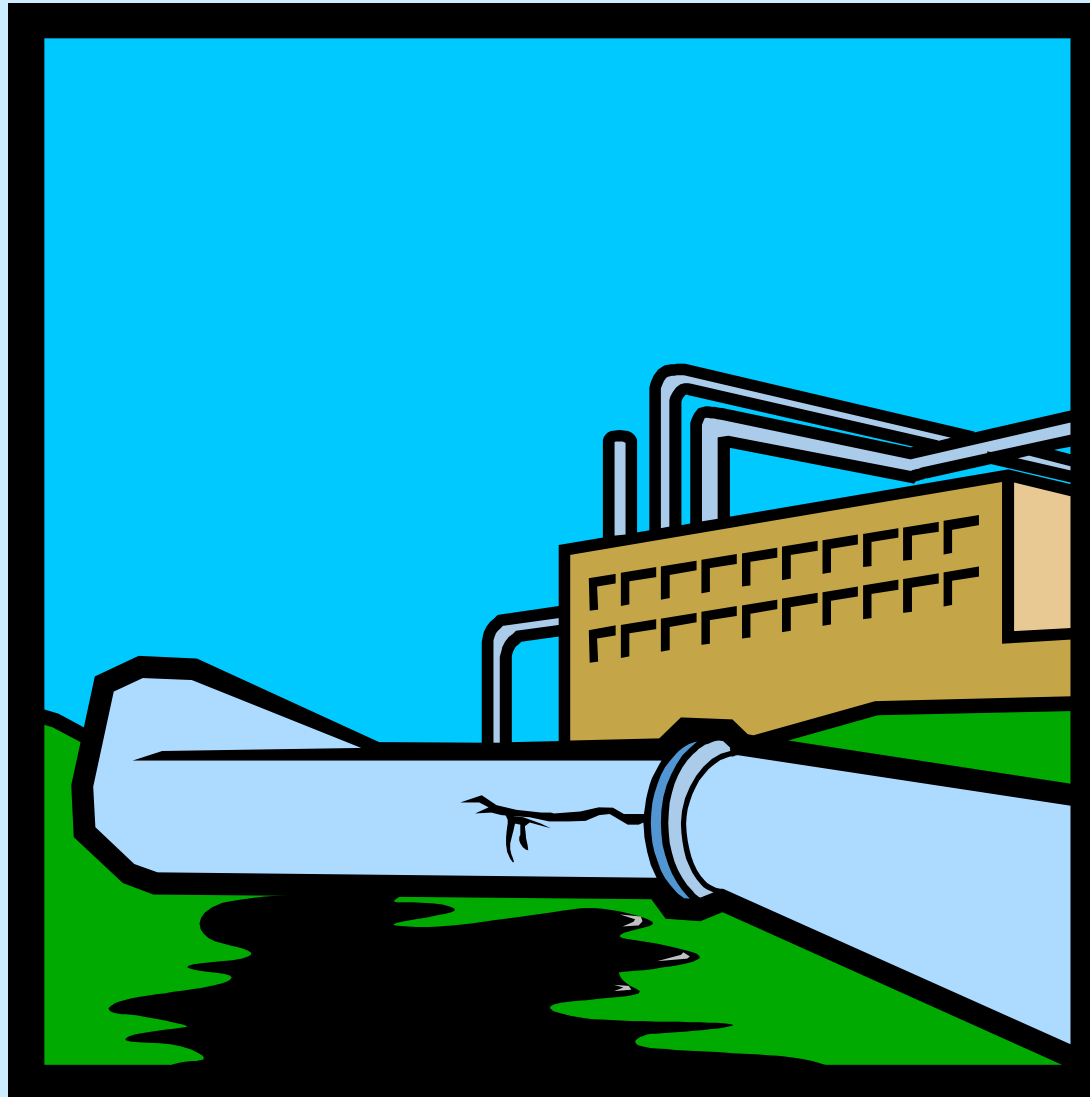
Pipelines

Decode	Fetch	Perform	Store	Decode	Fetch	Perform	Store

Decode	Fetch	Perform	Store

Decode	Fetch	Perform	Store

Decode	Fetch	Perform	Store

Decode	Fetch	Perform	Store

Decode	Fetch	Perform	Store

Analysis

- **Not pipelined**
  - **each instruction takes, say, 320 nanoseconds**
    - » **320 ns latency**
  - **3.125 billion instructions/second (GIPS)**

- **Pipelined**
  - **each instruction still takes 320 ns**
    - » **latency still 320 ns**
  - **an instruction completes every 80 ns**
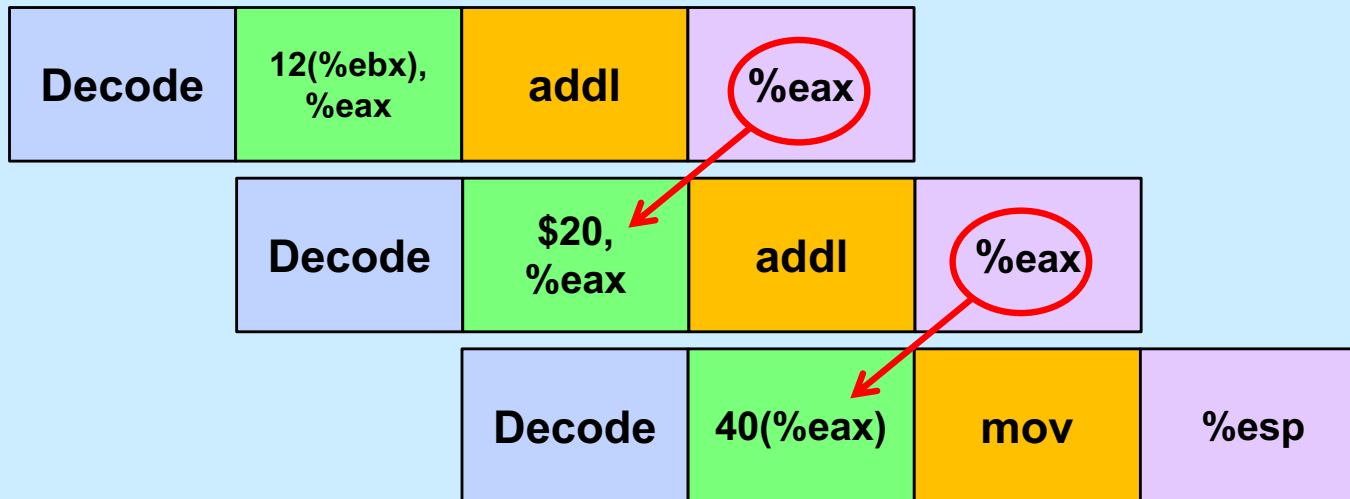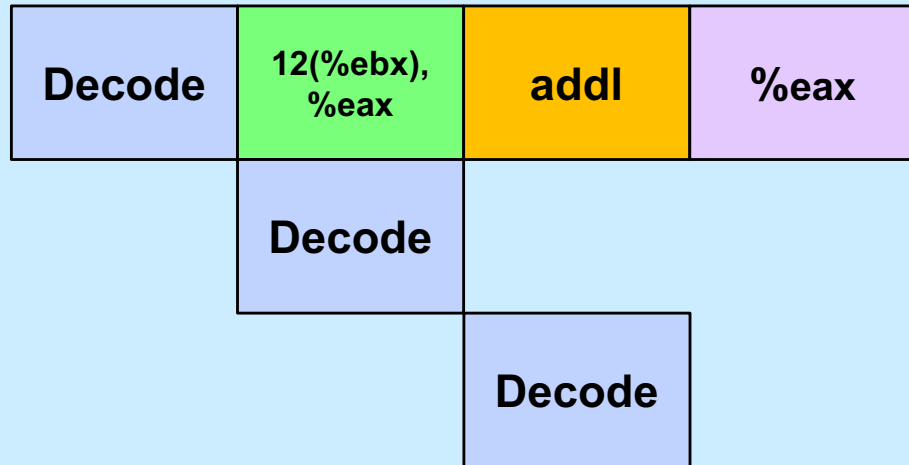   - » **12.5 GIPS throughput**

Hazards ...

Data Hazards

```
addl 12(%ebx), %eax
addl $20, %eax
movl 40(%eax), %esp
```

Decode	12(%ebx), %eax	addl	%eax

	Decode	$20, %eax	addl	%eax

		Decode	40(%eax)	mov	%esp

Coping

Decode	12(%ebx), %eax	addl	%eax

Decode

Decode

$20, %eax	addl	%eax

40(%eax)	mov

Control Hazards

```
  movl $0, %ecx
.L2:
  movl %edx, %eax
  andl $1, %eax
  addl %eax, %ecx
  shrl $1, %edx
  jne  .L2 # what goes in the pipeline?
  movl %ecx, %eax
  ...
```
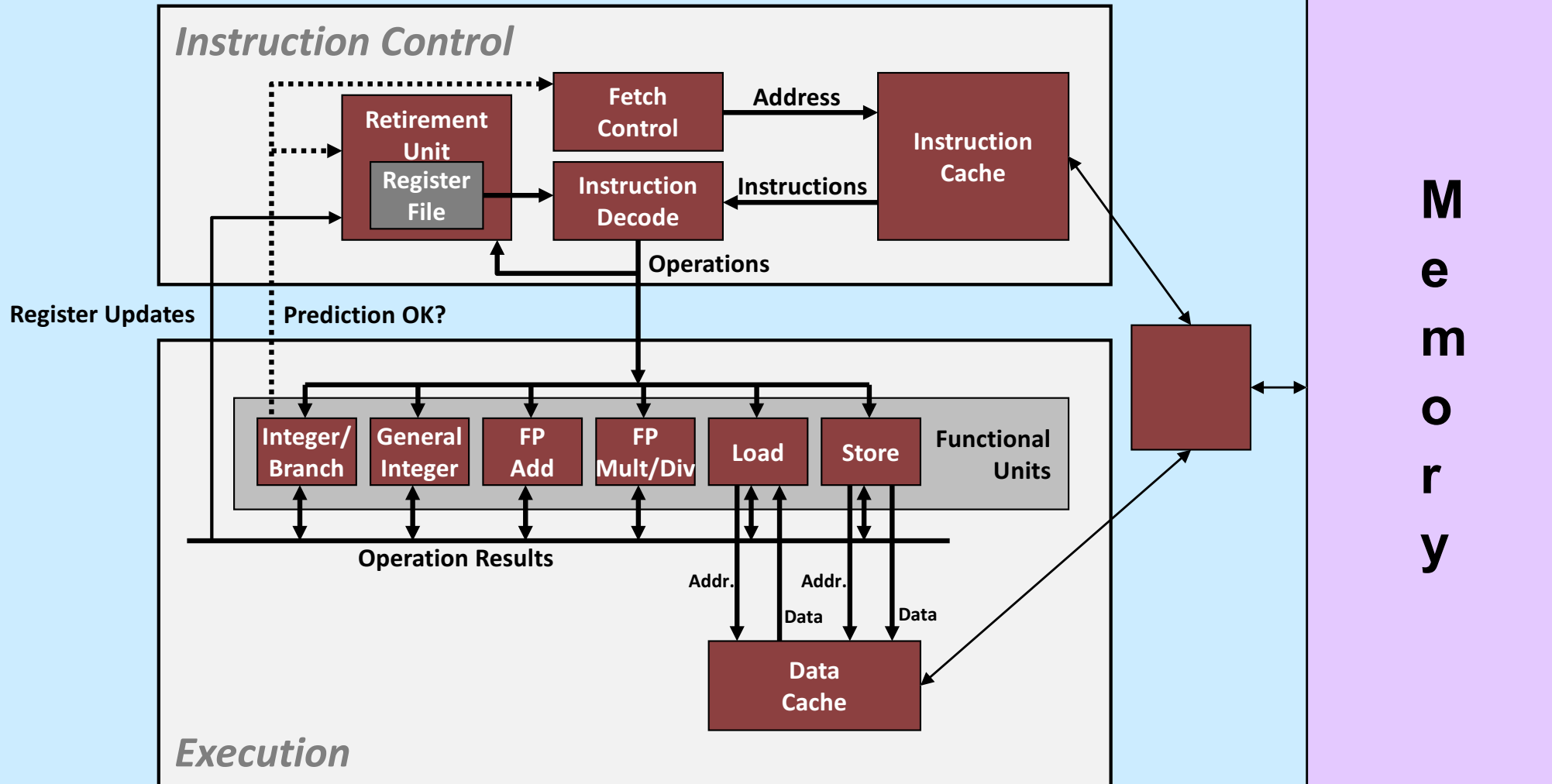
Coping: Guess ...

- **Branch prediction**
  - assume, for example, that conditional branches are always taken
  - but don't do anything to registers or memory until you know for sure

Modern CPU Design

Instruction Control

- Retirement Unit
  - Register File
- Fetch Control
- Instruction Decode
- Instruction Cache

Address

Instructions

Operations

Register Updates

Prediction OK?

Execution

Functional Units:
- Integer/ Branch
- General Integer
- FP Add
- FP Mult/Div
- Load
- Store

Operation Results

Addr.

Addr.

Data

Data

Data Cache

M e m o r y

Performance Realities

There's more to performance than asymptotic complexity

- **Constant factors matter too!**
  - easily see 10:1 performance range depending on how code is written
  - must optimize at multiple levels:
    - » algorithm, data representations, procedures, and loops
- **Must understand system to optimize performance**
  - how programs are compiled and executed
  - how to measure program performance and identify bottlenecks
  - how to improve performance without destroying code modularity and generality

Optimizing Compilers

- **Provide efficient mapping of program to machine**
  - **register allocation**
  - **code selection and ordering (scheduling)**
  - **dead code elimination**
  - **eliminating minor inefficiencies**
- **Don't (usually) improve asymptotic efficiency**
  - **up to programmer to select best overall algorithm**
  - **big-O savings are (often) more important than constant factors**
    - » **but constant factors also matter**
- **Have difficulty overcoming "optimization blockers"**
  - **potential memory aliasing**
  - **potential procedure side-effects**

Limitations of Optimizing Compilers

- **Operate under fundamental constraint**
  - must not cause any change in program behavior
  - often prevents it from making optimizations that would only affect behavior under pathological conditions
- **Behavior that may be obvious to the programmer can be obfuscated by languages and coding styles**
  - e.g., data ranges may be more limited than variable types suggest
- **Most analysis is performed only within procedures**
  - whole-program analysis is too expensive in most cases
- **Most analysis is based only on *static* information**
  - compiler has difficulty anticipating run-time inputs

- **When in doubt, the compiler must be conservative**

Generally Useful Optimizations

- **Optimizations that you or the compiler should do regardless of processor / compiler**

- **Code Motion**
  - **reduce frequency with which computation performed**
    - » **if it will always produce same result**
    - » **especially moving code out of loop**

```
void set_row(long *a, long *b,
    long i, long n){
  long j;
  for (j = 0; j < n; j++)
      a[n*i+j] = b[j];
}
```

→

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

Compiler-Generated Code Motion

```
void set_row(long *a, long *b,
   long i, long n){
   long j;
   for (j = 0; j < n; j++)
      a[n*i+j] = b[j];
}
```

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

```
set_row:
        testq    %rcx, %rcx           # Test n
        jle      .L1                  # If 0, goto done
        imulq    %rcx, %rdx           # i *= n
        leaq     (%rdi,%rdx,8), %rdi  # rowp = A + n*i*8
        movl     $0, %eax             # j = 0
.L3:                                  # loop:
        movq     (%rsi,%rax,8), %rdx  # t = b[j]
        movq     %rdx, (%rdi,%rax,8)  # rowp[j] = t
        addq     $1, %rax             # j++
        cmpq     %rcx, %rax           # Compare n:j
        jg       .L3                  # If >, goto loop
.L1:                                  # done:
        rep ; ret
```

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 Intel Nehalem, integer multiply requires 3 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;
}
```

Share Common Subexpressions

- Reuse portions of expressions
- Compilers often not very sophisticated in exploiting arithmetic properties

```
/* Sum neighbors of i,j */
up =     val[(i-1)*n + j  ];
down =  val[(i+1)*n + j  ];
left =  val[i*n      + j-1];
right = val[i*n      + j+1];
sum = up + down + left + right;
```

```
long inj = i*n + j;
up =     val[inj - n];
down =  val[inj + n];
left =  val[inj - 1];
right = val[inj + 1];
sum = up + down + left + right;
```

3 multiplications: i*n, (i−1)*n, (i+1)*n   **1 multiplication: i*n**

```
leaq    1(%rsi), %rax   # i+1
leaq    -1(%rsi), %r8   # i-1
imulq  %rcx, %rsi       # i*n
imulq  %rcx, %rax       # (i+1)*n
imulq  %rcx, %r8        # (i-1)*n
addq    %rdx, %rsi      # i*n+j
addq    %rdx, %rax      # (i+1)*n+j
addq    %rdx, %r8       # (i-1)*n+j
```

```
imulq    %rcx, %rsi  # i*n
addq     %rdx, %rsi  # i*n+j
movq     %rsi, %rax  # i*n+j
subq     %rcx, %rax  # i*n+j-n
leaq     (%rsi,%rcx), %rcx # i*n+j+n
```

Quiz 1

The fastest means for evaluating

```
n*n + 2*n + 1
```

requires exactly:

a) 2 multiplies and 2 additions

b) one multiply and two additions

c) one multiply and one addition

d) three additions
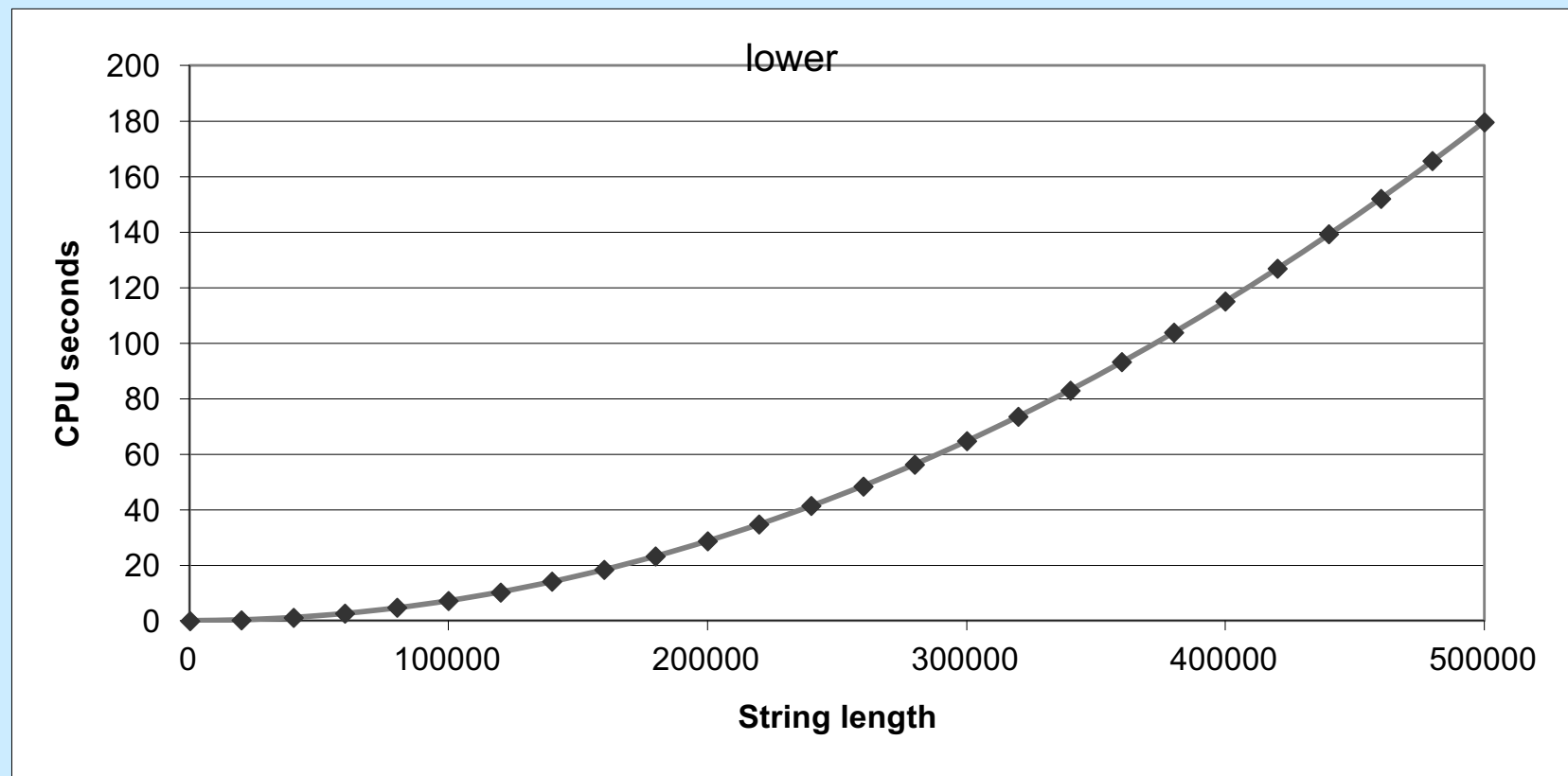
Optimization Blocker #1: Procedure Calls

- **Procedure to convert string to lower case**

```c
void lower(char *s){
  int i;
  for (i = 0; i < strlen(s); i++)
    if (s[i] >= 'A' && s[i] <= 'Z')
      s[i] -= ('A' - 'a');
}
```

Lower Case Conversion Performance

- **Time quadruples when string length doubles**
- **Quadratic performance**

Convert Loop To Goto Form

```c
void lower(char *s){
    int i = 0;
    if (i >= strlen(s))
        goto done;
 loop:
    if (s[i] >= 'A' && s[i] <= 'Z')
        s[i] -= ('A' - 'a');
    i++;
    if (i < strlen(s))
        goto loop;
 done:
}
```

- **strlen executed every iteration**

Calling Strlen

```c
size_t strlen(const char *s){
    size_t length = 0;
    while (*s != '\0') {
        s++;
        length++;
    }
    return length;
}
```

- **strlen performance**
  - only way to determine length of string is to scan its entire length, looking for null character
- **Overall performance, string of length N**
  - N calls to strlen
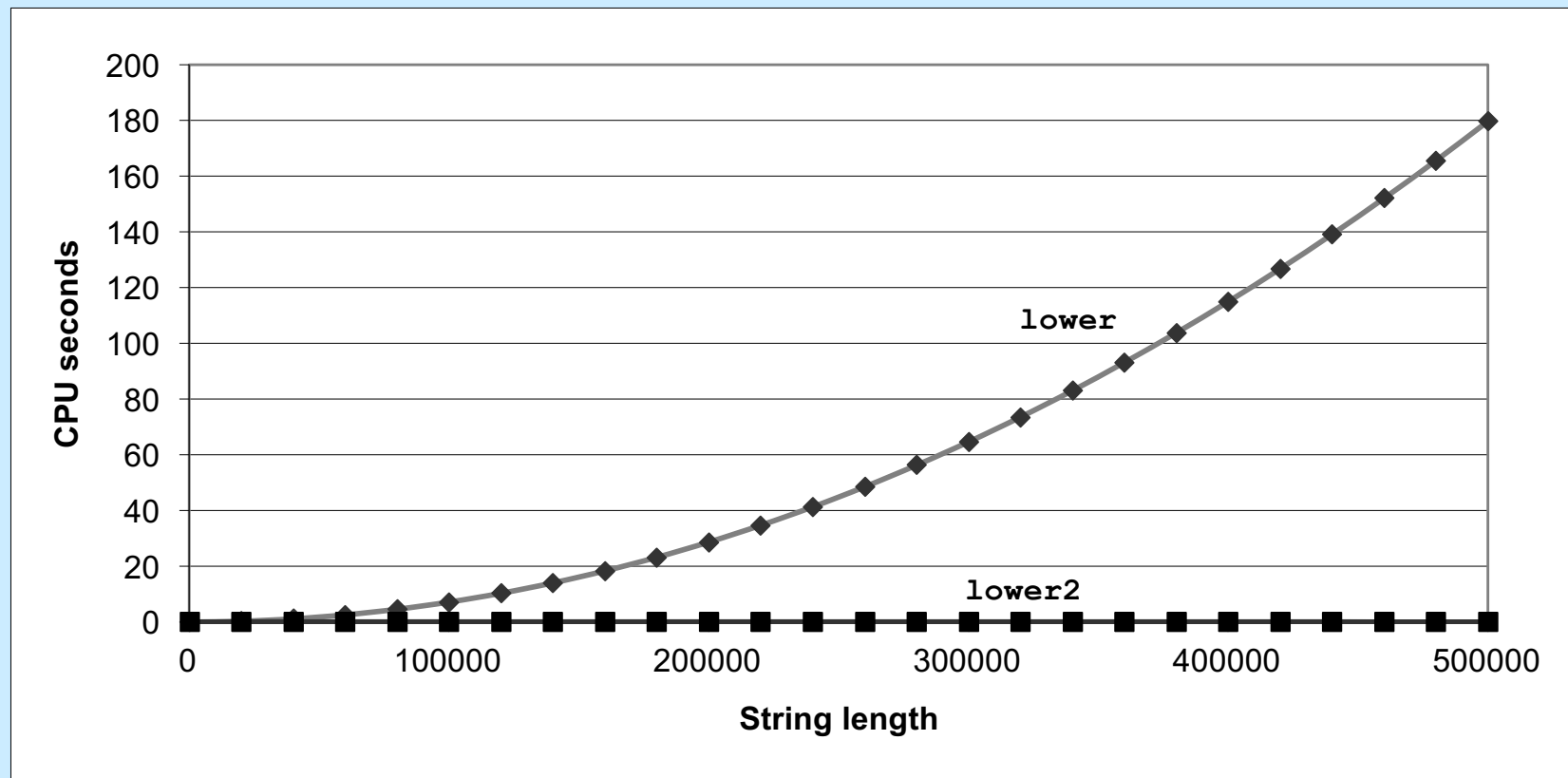 - overall $O(N^2)$ performance

Improving Performance

```
void lower2(char *s){
    int i;
    int len = strlen(s);
    for (i = 0; i < len; i++)
        if (s[i] >= 'A' && s[i] <= 'Z')
            s[i] -= ('A' - 'a');
}
```

- **Move call to `strlen` outside of loop**
  - since result does not change from one iteration to another
  - form of code motion

Lower-Case Conversion Performance

- ## Time doubles when string-length doubles
  - – **linear performance of lower2**

Optimization Blocker: Procedure Calls

- *Why couldn't compiler move* `strlen` *out of inner loop?*
  - **procedure may have side effects**
    - » **alters global state each time called**
  - **function may not return same value for given arguments**
    - » **depends on other parts of global state**
    - » **procedure `lower` could interact with `strlen`**

- **Warning:**
  - **compiler treats procedure call as a black box**
  - **weak optimizations near them**

- **Remedies:**
  - **use of `inline` functions**
    - » **gcc does this with –O2**
  - **do your own code motion**

```
int lencnt = 0;
size_t strlen(const char *s){
    size_t length = 0;
    while (*s != '\0') {
        s++; length++;
    }
    lencnt += length;
    return length;
}
```

Memory Matters

```c
/* Sum rows of n X n matrix a
   and store result in vector b  */
void sum_rows1(long *a, long *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i*n + j];
    }
}
```

```
# sum_rows1 inner loop
.L3:
    movq    (%rdi), %rcx        # rcx = *a
    addq    %rcx, (%rsi,%rax,8)  # b[i] += rcx
    addq    $8, %rdi            # a++
    cmpq    %r8, %rdi
    jne     .L3
```

- **Code updates b[i] on every iteration**
- **Why couldn't compiler optimize this away?**

Memory Aliasing

```
/* Sum rows of n X n matrix a
   and store result in vector b  */
void sum_rows1(int *a, int *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i*n + j];
    }
}
```

```
int A[9] =
   { 0,    1,    2,
     4,    8,   16,
    32,   64, 128};

int *B = &A[3];

sum_rows1(A, B, 3);
```

Value of B:

init:	[4, 8, 16]
i = 0:	[3, 8, 16]
i = 1:	[3, 22, 16]
i = 2:	[3, 22, 224]

- **Code updates `b[i]` on every iteration**

- **Must consider possibility that these updates will affect program behavior**

Removing Aliasing

```
/* Sum rows of n X n matrix a
   and store result in vector b  */
void sum_rows2(int *a, int *b, int n) {
    long i, j;
    for (i = 0; i < n; i++) {
        int val = 0;
        for (j = 0; j < n; j++)
            val += a[i*n + j];
        b[i] = val;
    }
}
```

```
# sum_rows2 inner loop
.L4:
    addq    (%rdi), %rax
    addq    $8, %rdi
    cmpq    %rcx, %rdi
    jne     .L4
```

- **No need to store intermediate results**

Optimization Blocker: Memory Aliasing

- **Aliasing**
  - **two different memory references specify single location**
  - **easy to have happen in C**
    - » **since allowed to do address arithmetic**
    - » **direct access to storage structures**
  - **get in habit of introducing local variables**
    - » **accumulating within loops**
    - » **your way of telling compiler not to check for aliasing**

C99 to the Rescue

- ## New attribute
  - ### restrict
    - » applied to a pointer, tells the compiler that the object pointed to will be accessed only via this pointer
    - » compiler thus doesn't have to worry about aliasing
    - » but the programmer does ...
    - » syntax

      ```
      int *restrict pointer;
      ```

Memory Matters, Fixed

```
/* Sum rows of n X n matrix a
   and store result in vector b  */
void sum_rows3(long *restrict a, long *restrict b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i*n + j];
    }
}
```

```
# sum_rows1 inner loop
.L3:
    addq    (%rdi), %rax
    addq    $8, %rdi
    cmpq    %rcx, %rdi
    jne     .L3
```

- **Code doesn't update `b[i]` on every iteration**

Exploiting Instruction-Level Parallelism

- **Need general understanding of modern processor design**
  - hardware can execute multiple instructions in parallel
- **Performance limited by data dependencies**
- **Simple transformations can have dramatic performance improvement**
  - compilers often cannot make these transformations
  - lack of associativity and distributivity in floating-point arithmetic

Benchmark Example: Datatype for Vectors

```
/* data structure for vectors */
typedef struct{
    int len;
    data_t *data;
} vec_t, *vec_ptr_t;
```
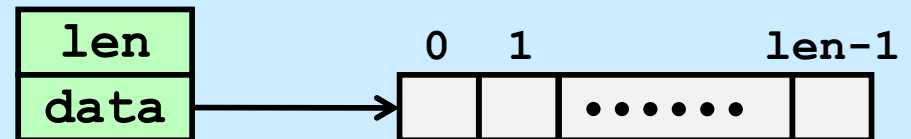


```
/* retrieve vector element and store at val */
int get_vec_element(vec_ptr_t v, int idx, data_t *val){
    if (idx < 0 || idx >= v->len)
        return 0;
    *val = v->data[idx];
    return 1;
}

/* return length of vector */
int vec_length(vec_ptr_t v) {
    return v->len;
}
```

Benchmark Computation

```
void combine1(vec_ptr_t v, data_t *dest){
    long int i;
    *dest = IDENT;
    for (i = 0; i < vec_length(v); i++) {
        data_t val;
        get_vec_element(v, i, &val);
        *dest = *dest OP val;
    }
}
```

Compute sum or product of vector elements
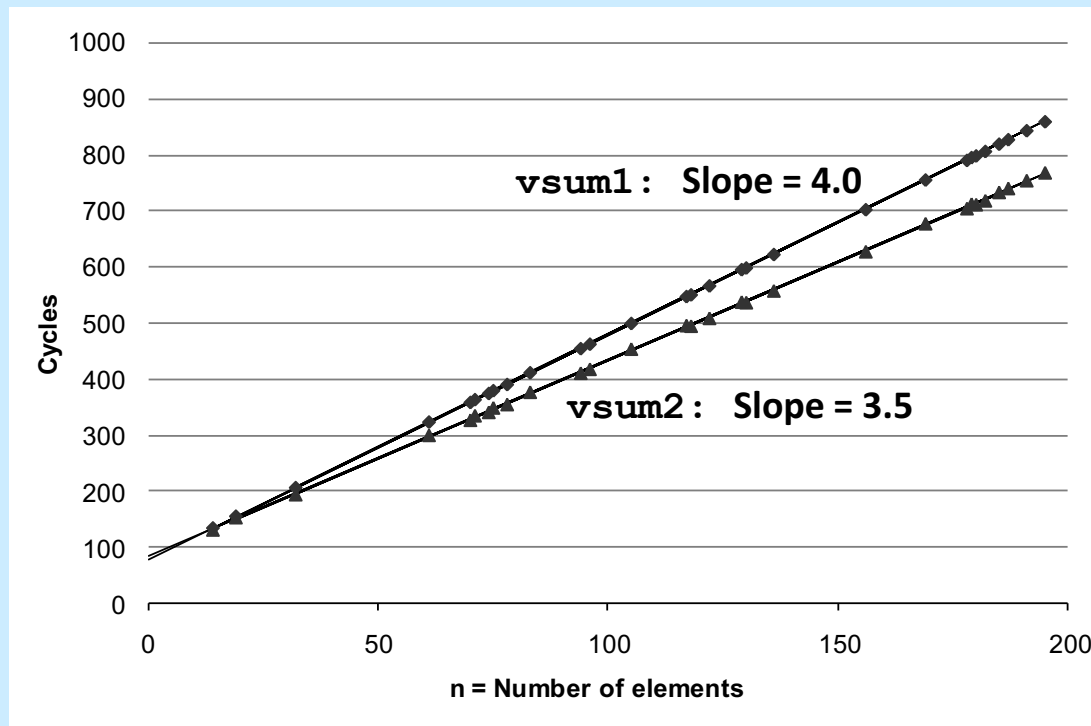
- **Data Types**
  - **use different declarations for `data_t`**
    - » **`int`**
    - » **`float`**
    - » **`double`**

- **Operations**
  - **use different definitions of `OP` and `IDENT`**
    - » **`+, 0`**
    - » **`*, 1`**

Cycles Per Element (CPE)

- **Convenient way to express performance of program that operates on vectors or lists**
- **Length = n**
- **T = CPE*n + Overhead**
  - **CPE is slope of line**

vsum1: Slope = 4.0

vsum2: Slope = 3.5

Cycles (y-axis): 0, 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000

n = Number of elements (x-axis): 0, 50, 100, 150, 200

Benchmark Performance

```
void combine1(vec_ptr_t v, data_t *dest){
    long int i;
    *dest = IDENT;
    for (i = 0; i < vec_length(v); i++) {
        data_t val;
        get_vec_element(v, i, &val);
        *dest = *dest OP val;
    }
}
```

Compute sum or product of vector elements

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine1 unoptimized	29.0	29.2	27.4	27.9
Combine1 –O1	12.0	12.0	12.0	13.0

Move vec_length

```
void combine2(vec_ptr_t v, data_t *dest){
    long int i;
    long int length = vec_length(v);
    *dest = IDENT;
    for (i = 0; i < length; i++) {
        data_t val;
        get_vec_element(v, i, &val);
        *dest = *dest OP val;
    }
}
```

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine1 unoptimized	29.0	29.2	27.4	27.9
Combine1 –O1	12.0	12.0	12.0	13.0
Combine2	8.03	8.09	10.09	12.08

Eliminate Procedure Calls

```c
void combine3(vec_ptr_t v, data_t *dest){
    long int i;
    long int length = vec_length(v);
    data_t *data = get_vec_start(v);
    *dest = IDENT;
    for (i = 0; i < length; i++) {
        *dest = *dest OP data[i];
    }
}
```

```c
data_t *get_vec_start(
    vec_ptr v) {
  return v->data;
}
```

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine2	8.03	8.09	10.09	12.08
Combine3	6.01	8.01	10.01	12.02

Eliminate Unneeded Memory References

```
void combine4(vec_ptr_t v, data_t *dest){
    int i;
    int length = vec_length(v);
    data_t *d = get_vec_start(v);
    data_t t = IDENT;
    for (i = 0; i < length; i++)
        t = t OP d[i];
    *dest = t;
}
```

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine1 –O1	12.0	12.0	12.0	13.0
Combine4	2.0	3.0	3.0	5.0

Quiz 2

Combine4 is pretty fast; we've done all the "obvious" optimizations. How much faster will we be able to make it? (Hint: it involves taking advantage of pipelining and multiple functional units on the chip.)

a) 1× (it's already as fast as possible)

b) 2× – 4×

c) 16× – 64×