HARVARD
School of Engineering
and Applied Sciences

Program optimization

CS61, Lecture 7

Prof. Stephen Chong

September 23, 2010

Announcements

- Lab 2 is due in one week: Thurs 30 Sep
- Note about in-section quizzes
  - From now on they will be closed book
  - They will be brief, around 10-15 minutes.
    - No strict time limit, not intended to take long to complete
  - Remember: two lowest scores dropped
- Brian Kernighan: co-inventor of C
  - Talk at 4pm
- There will be some minor changes to the cs61.seas server in the next few days
  - Keep an eye on the CS 61 website for more info

Today

- Program optimization
  - Overview
  - Code motion
  - Strength reduction
  - Common subexpressions
  - Optimization blockers
    - Procedure calls
    - Aliasing
  - Understanding modern processors
  - Loop unrolling
  - Summary

Getting the best performance

- There's more to performance than asymptotic complexity!
- Constant factors matter too
  - Easily see 10×–100× difference depending on how code is written
  - Must optimize at multiple levels:
    - algorithm structure (locality, instruction level parallelism, ...)
    - data representations (e.g., structs vs arrays)
    - coding style (e.g., unnecessary procedure calls, unrolling, reordering, ...)
- Must understand underlying system to optimize performance
  - How programs are compiled and executed
  - How to measure program performance and identify bottlenecks
  - How to improve performance while maintaining code modularity and generality

Optimizing compilers (e.g., gcc)

- Compilers do a **lot** of optimization when generating machine code

- Use optimization flags when compiling
  - Default is no optimization (-O0)
  - Good choices for gcc: -O2, -O3, -march=xxx, -m64
  - Try different flags and maybe different compilers

Optimizing compilers (e.g., gcc)

- Compilers are **good** at: mapping program to machine
  - register allocation
  - instruction selection and ordering (scheduling)
  - dead code elimination
  - eliminating minor inefficiencies
- Compilers are **not good** at: improving 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
- Compilers are **not good** at: overcoming "optimization blockers"
  - potential memory aliasing
  - potential procedure side-effects

Limitations of Optimizing Compilers

- When in doubt, the **compiler must be conservative**
- Must not change program behavior under any possible condition
  - Often prevents it from making optimizations when 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
  - Not amenable to modular compilation
- Code analysis generally based only on **static** information
  - That is, whatever it can determine at compile time
  - Difficult (in general, undecidable) to determine run-time, or **dynamic**, behavior

Machine-independent optimizations

- Some simple optimizations, regardless of specific machine or compiler
  - Code motion
  - Strength reduction
  - Common subexpressions
- For some instances of these optimizations, almost all compilers will perform them
- For other instances, very difficult for a compiler to perform them
  - You need to understand why

Code motion

- **Key idea:** Move code to reduce the number of times it executes
- Most common case: move code out of loop
- E.g.

```
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];
    }
}
```

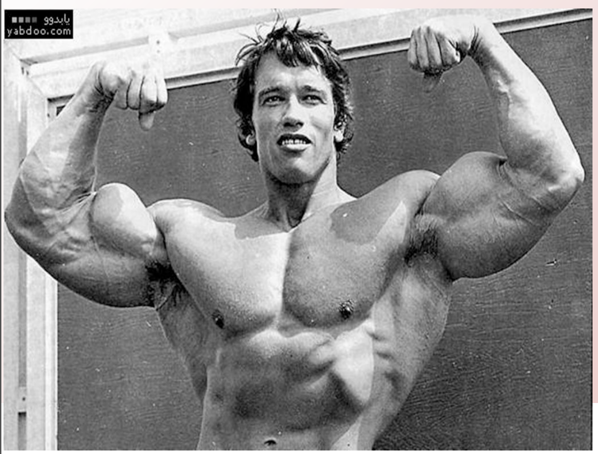
```
    long j;
    int ni = n*i;
    for (j = 0; j < n; j++) {
        a[ni+j] = b[j];
    }
```

  - Moving code means n-1 fewer multiplications!

Compiler generated code motion

```
set_row:
        pushl   %ebp                    # Setup
        movl    %esp, %ebp
        pushl   %esi
        pushl   %ebx
        movl    12(%ebp), %esi          # esi = b
        movl    20(%ebp), %ebx          # ebx = n
        testl   %ebx, %ebx              # is n <= 0?
        jle     .L26                    #   return
        movl    %ebx, %edx              # edx = n
        imull   16(%ebp), %edx          # edx = n*i
        movl    8(%ebp), %eax           # eax = a
        leal    (%eax,%edx,4), %edx # edx = &(a[n*i])
        movl    $0, %ecx                # ecx = 0
.L25:
        movl    (%esi,%ecx,4), %eax # eax = &(b[j])
        movl    %eax, (%edx)            # a[n*i+j] = b[j]
        addl    $1, %ecx                # j++
        addl    $4, %edx                # edx = next element of a
        cmpl    %ecx, %ebx              # j == n?
        jne     .L25                    #   if not, continue loop
.L26:
        popl    %ebx                   # Finish
        popl    %esi
        popl    %ebp
        ret
```

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



Strength reduction

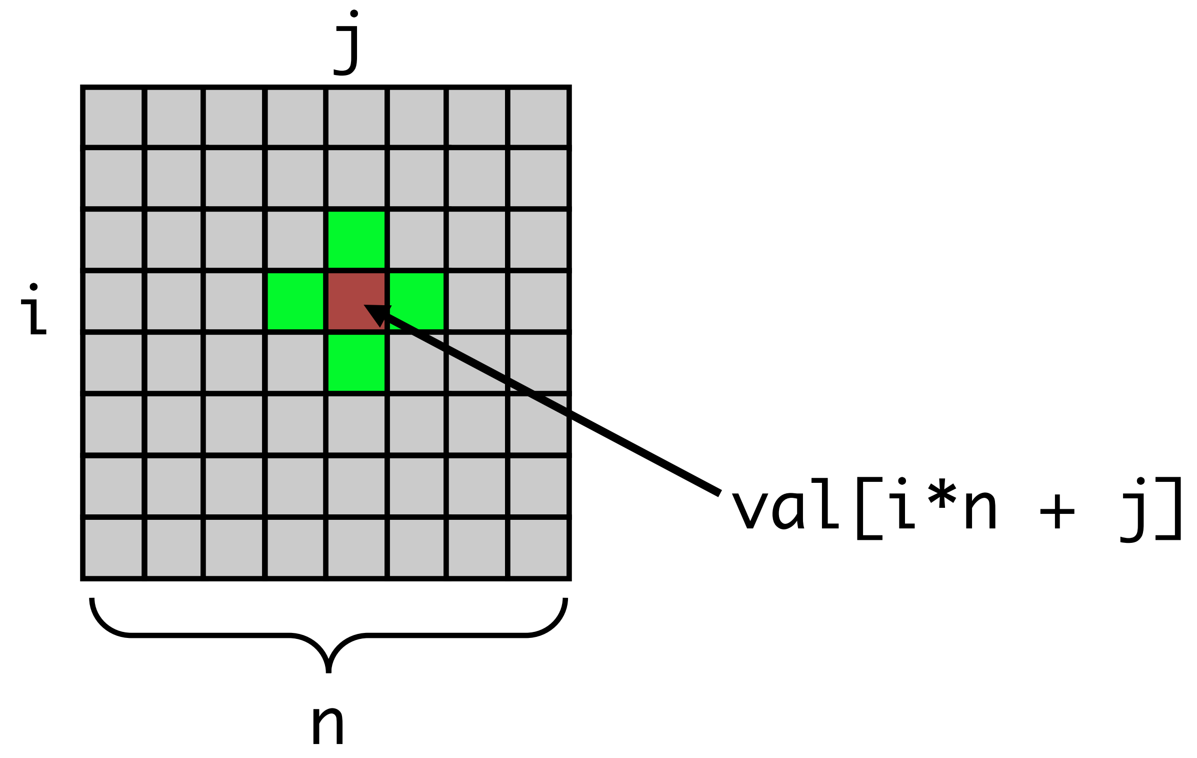
- **Key idea:** replace expensive operations with cheaper ones
- E.g., reduce a multiplication inside a loop to an addition
    - Addition of integers much faster than multiplication

```
/* sum column i of n x n array a */
int sum_col(int *a, int n, int i) {
    int s = 0;
    for (j = 0; j < n; j++) {
        s += a[n*j+i];
    }
    return s;
}
```

```
/* sum column i of n x n array a */
int sum_col(int *a, int n, int i) {
    int s = 0;
    int r = 0;
    for (j = 0; j < n; j++) {
        s += a[r+i];
        r += n;
    }
    return s;
}
```

Share Common Subexpressions

- **Key idea:** reuse common 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;
```

```
int 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;
```

Share Common Subexpressions

```
imull   %eax, %esi
leal    (%ebx,%esi), %esi
leal    -1(%ecx), %edx
imull   %eax, %edx
addl    %ebx, %edx
addl    $1, %ecx
imull   %ecx, %eax
addl    %eax, %ebx
movl    (%edi,%ebx,4), %eax
addl    (%edi,%edx,4), %eax
movl    -4(%edi,%esi,4), %edx
addl    4(%edi,%esi,4), %edx
addl    %edx, %eax
```

3 multiplications

```
imull   %ecx, %edx
addl    16(%ebp), %edx
leal    0(,%edx,4), %edi
movl    %edx, %esi
subl    %ecx, %esi
movl    -4(%ebx,%edi), %eax
addl    (%ebx,%esi,4), %eax
addl    %edx, %ecx
movl    4(%ebx,%edi), %edx
addl    (%ebx,%ecx,4), %edx
addl    %edx, %eax
```

1 multiplication

```
/* 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;
```

```
int 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;
```

Today

- Program optimization
  - Overview
  - Code motion
  - Strength reduction
  - Common subexpressions
  - Optimization blockers
    - Procedure calls
    - Aliasing
  - Understanding modern processors
  - Loop unrolling
  - Summary

Optimization Blocker: Procedure Calls

- Converting a string to lower case:

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

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

- What's wrong (performance-wise) with this code?

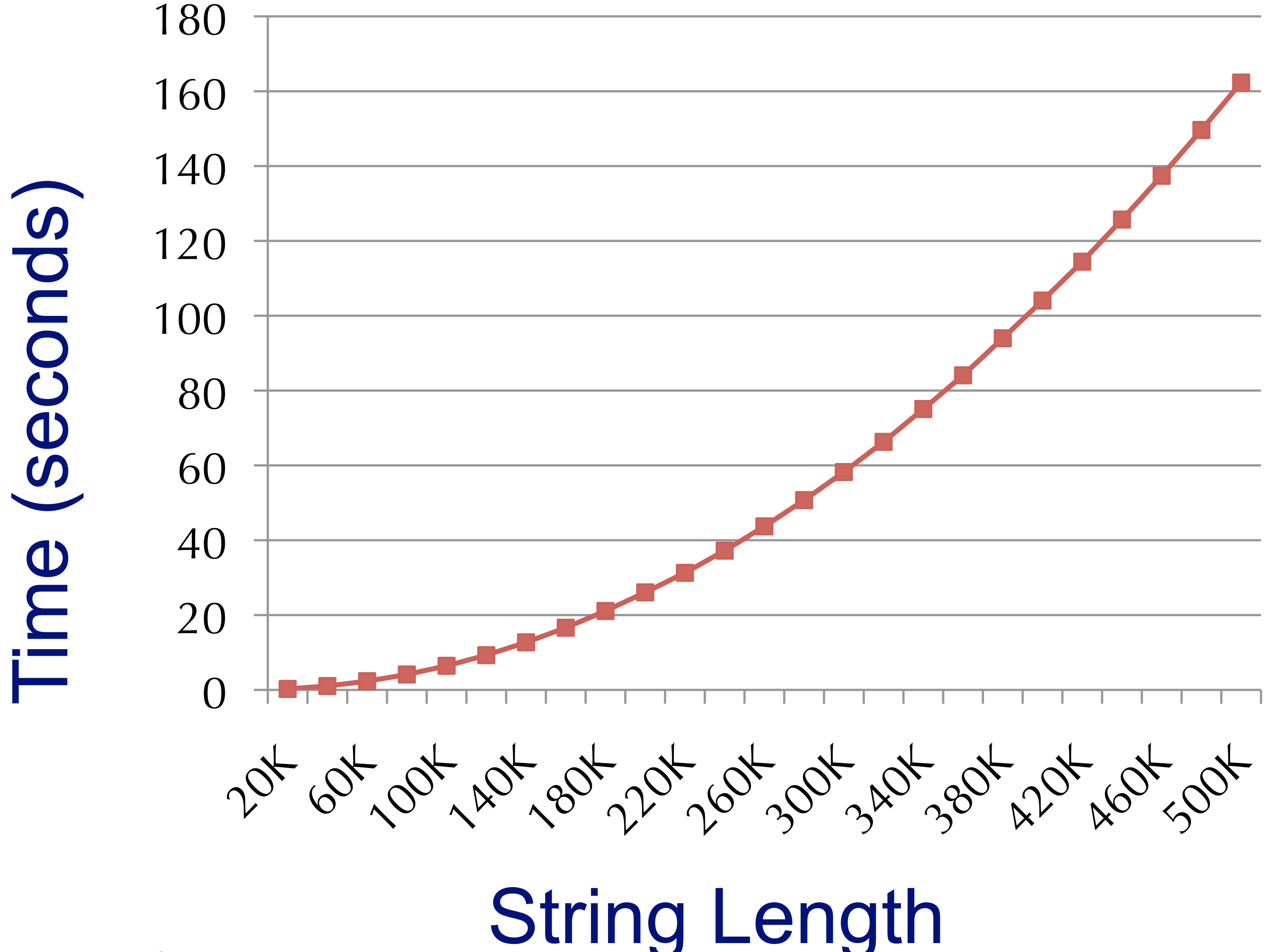
Convert Loop To Goto Form

- `strlen` executed every iteration!
- `strlen()` performance
  - Must scan string looking for null character.
- Overall performance, string of length *n*
  - *n* calls to `strlen`
  - Require times
    n, *n*-1, *n*-2, …, 1
  - Overall O(n^2) performance

```
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:
}
```

Lower Case Conversion Performance

- $O(n^2)$
  - Quadratic performance
  - Time quadruples when we double the input string length



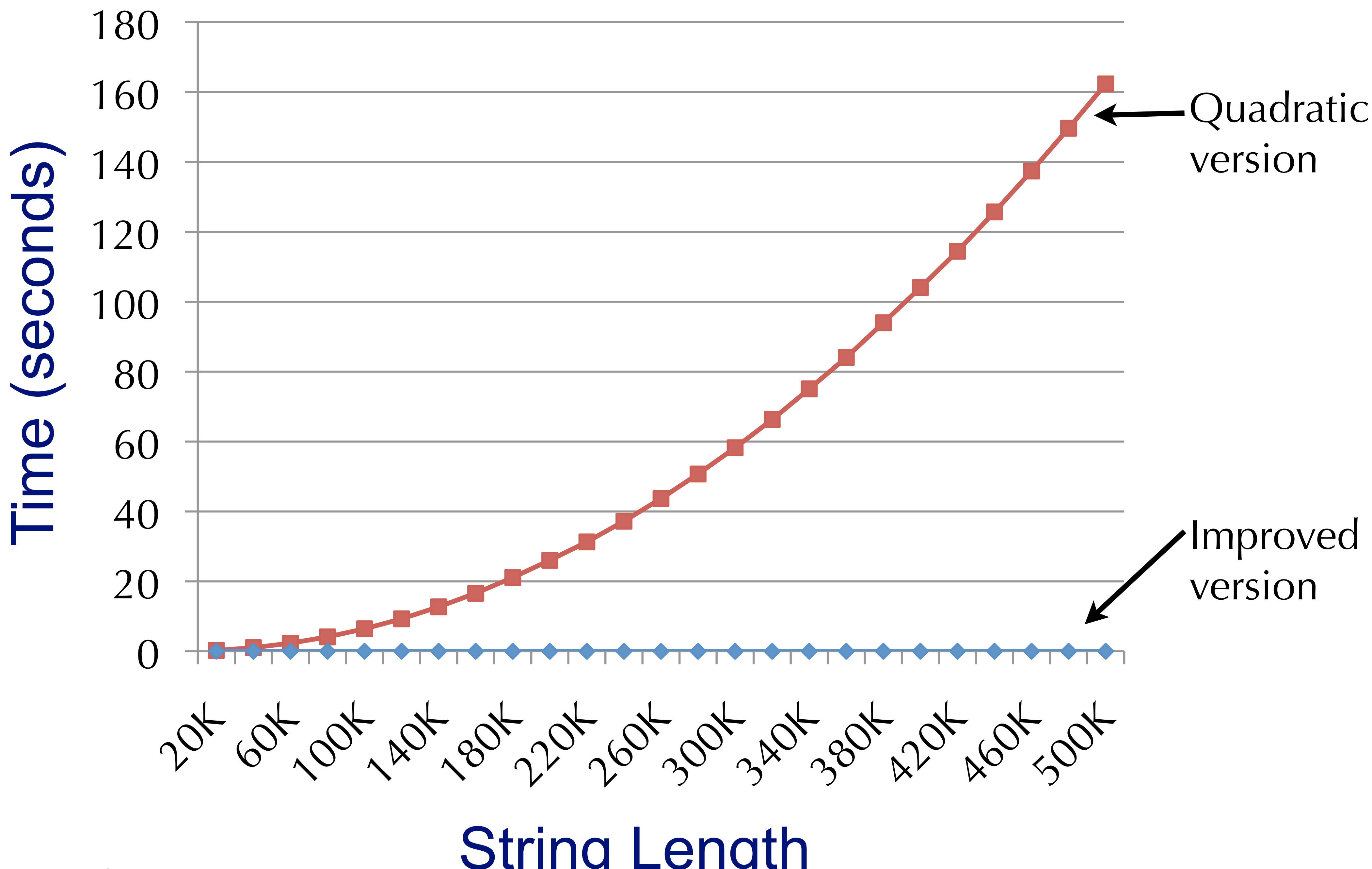
How to improve performance?

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

- Code motion!
  - Move call to `strlen()` outside of loop
  - OK because result does not change from one iteration to another

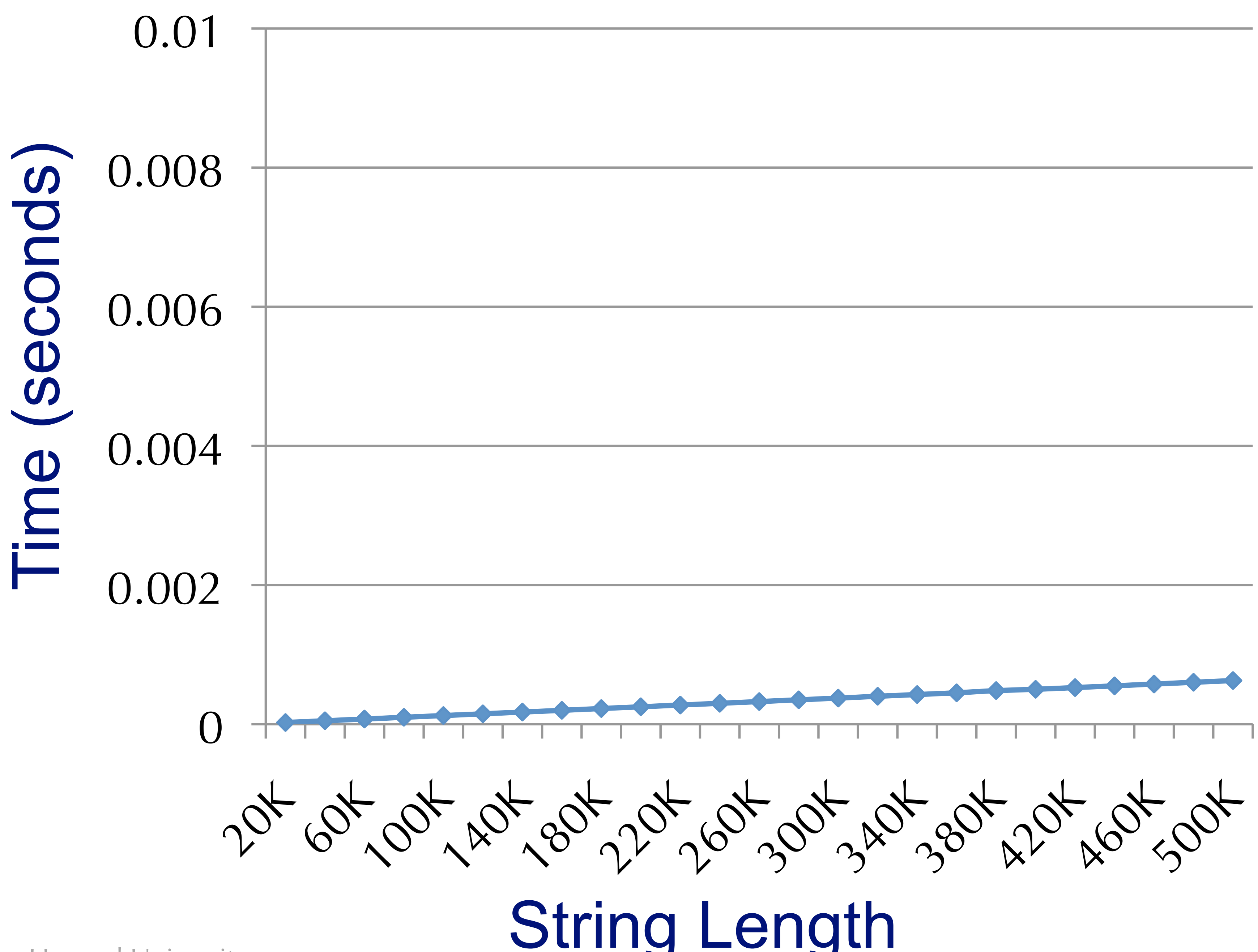
```
void lower(char *s)
{
  int i;
  int len = strlen(s);
  for (i = 0; i < len; i++)
    if (s[i] >= 'A' && s[i] <= 'Z')
      s[i] -= ('A' - 'a');
}
```

Improved performance!



Improved performance!

- Linear performance



Optimization Blocker: Procedure Calls

- Why couldn't compiler move `strlen()` out of inner loop?
- The compiler treats procedure calls as a "black box"
  - Must be conservative!
- Procedure may be **nondeterministic**
  - Does not return same value each time it is called with same inputs
  - Output could depend on global state (not just its input parameters)
- Procedure may have **side effects**
  - Alters global state each time called

Example: strlen with side effects

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

- Calling strlen once versus calling it *n* times has different behavior!

Potential remedies

- Do your own code motion
  - Rewrite code to move procedure call outside of the inner loop
- Use the `inline` keyword
  - Tells compiler that the function code can be inserted into the calling function
  - Allows compiler to optimize across caller and callee
  - Also done by default (for "simple" functions) when using `gcc -O3` (or use `-finline-functions`)

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

Optimization blocker: aliasing

```
void twiddle1(int *xp, int *yp) {
    *xp += *yp;
    *xp += *yp;
}
```

```
void twiddle2(int *xp, int *yp) {
    *xp += 2* *yp;

}
```

- Are the two functions above equivalent?
  - If so, `twiddle2` looks more efficient. Compiler should optimize `twiddle1` so it looks like `twiddle2`, right?

Optimization blocker: aliasing

```
void twiddle1(int *xp, int *yp) {
    *xp += *yp;
    *xp += *yp;
}
```

```
void twiddle2(int *xp, int *yp) {
    *xp += 2* *yp;

}
```

- But what if `xp` and `yp` are equal?
  - e.g., `int foo = 42; twiddle1(&foo, &foo);`
  - `twiddle1` computes:
    - ```
      foo += foo;    // doubles foo
      foo += foo;    // doubles foo again
      ```
  - `twiddle2` computes:
    - `foo += 2* foo; // triples foo`
- Not equivalent!!!
      ```

Memory aliasing

- If two pointers point to the same memory location, they *alias* each other.
- Compiler must assume that pointers may alias each other
  - Must be conservative!
  - Severely limits optimizations
- Lesson: Reduce unnecessary memory accesses

Reduce unnecessary memory accesses

- The following programs are not equivalent
  - Why?
- `prod_array1` must access memory repeatedly
  - Compiler cannot remove these accesses
- `prod_array2` can be compiled using a register for `res`
  - Much more efficient

```
void prod_array1(int *a, int n,
                 int *dest) {
  int i;
  for (i = 0; i < n; i++) {
    *dest = *dest * a[i];
  }
}
```

```
void prod_array2(int *a, int n,
                 int *dest) {
  int i, res = 1;
  for (i = 0; i < n; i++) {
    res = res * a[i];
  }
  *dest = res;
}
```

Today

- Program optimization
  - Overview
  - Code motion
  - Strength reduction
  - Common subexpressions
  - Optimization blockers
    - Procedure calls
    - Aliasing
  - Understanding modern processors
  - Loop unrolling
  - Summary

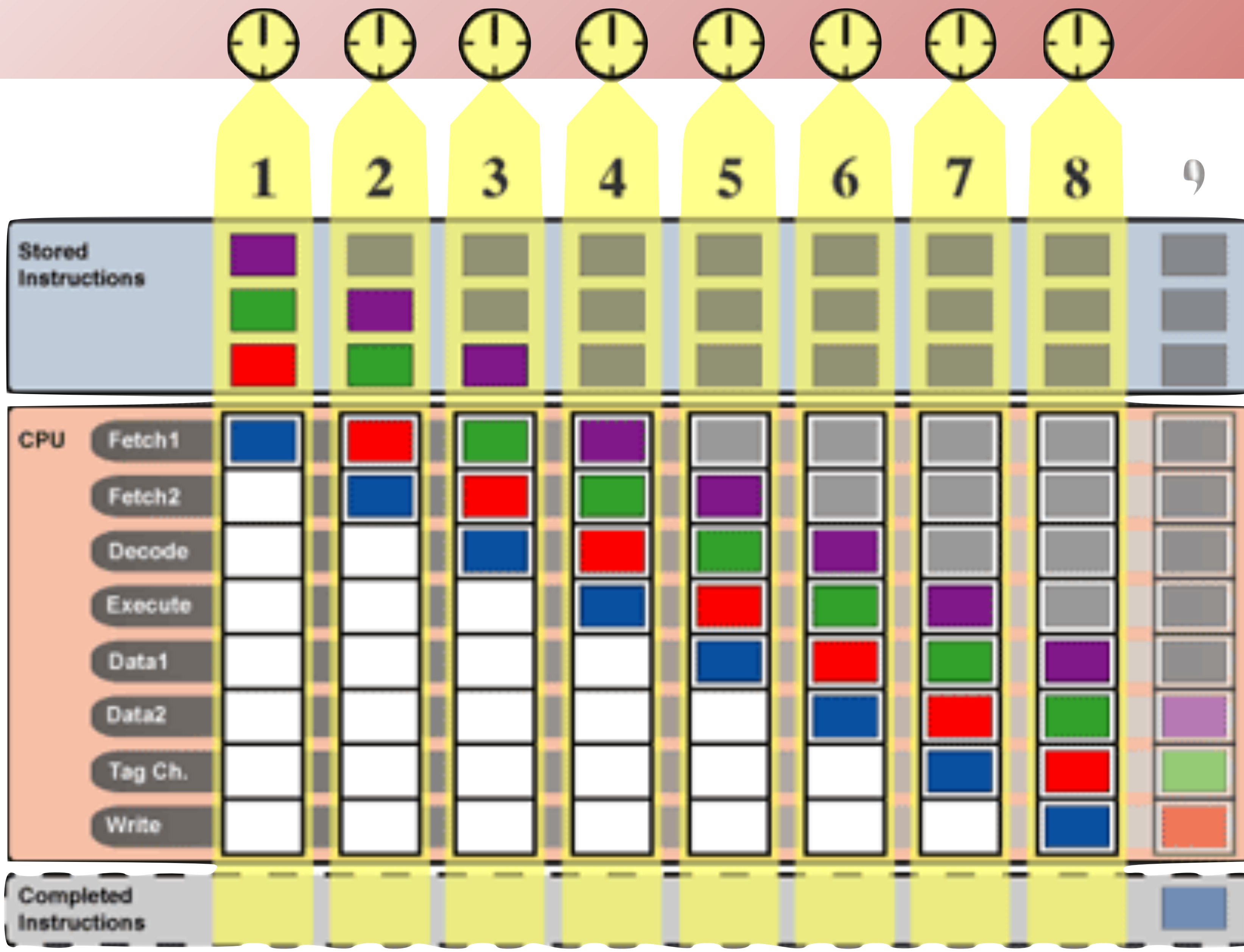
Three kinds of parallelism

- Three kinds of parallelism supported by modern CPUs:
  - Pipelining
  - Superscalar
  - Multicore

Pipelining

- Executing an instruction involves different stages
  - Fetch instruction from memory
  - Decode instruction
  - Load operands from memory (if necessary)
  - Perform operation (e.g., add, multiply, etc.) and update registers
  - Store results to memory (if necessary)
- "Classic" view of a CPU: Processor does **one thing at a time**
- Different hardware used for each stage
  - Can start fetching next instruction while previous instruction executing
  - Stages form a "pipeline" that instructions move down

Pipelining



http://arstechnica.com/old/content/2004/09/pipelining-2.ars/4

- A given instruction may take 8 cycles to complete (**latency**) but the processor can complete one instruction per cycle (**throughput**)!

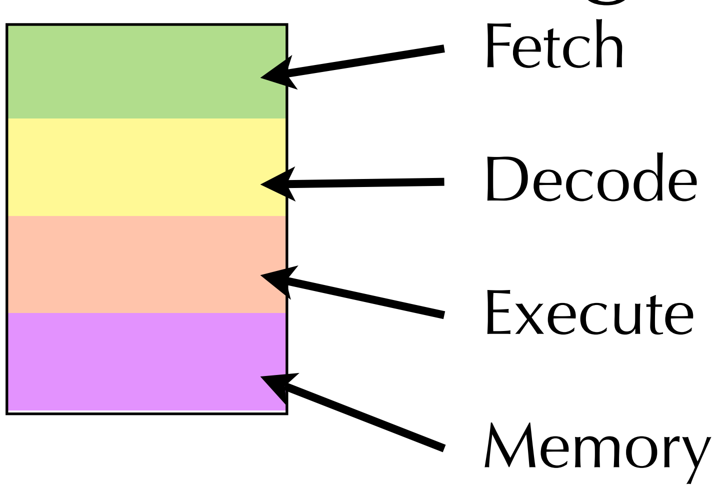
Exploiting pipelining

- Consider simple example: iterating over array

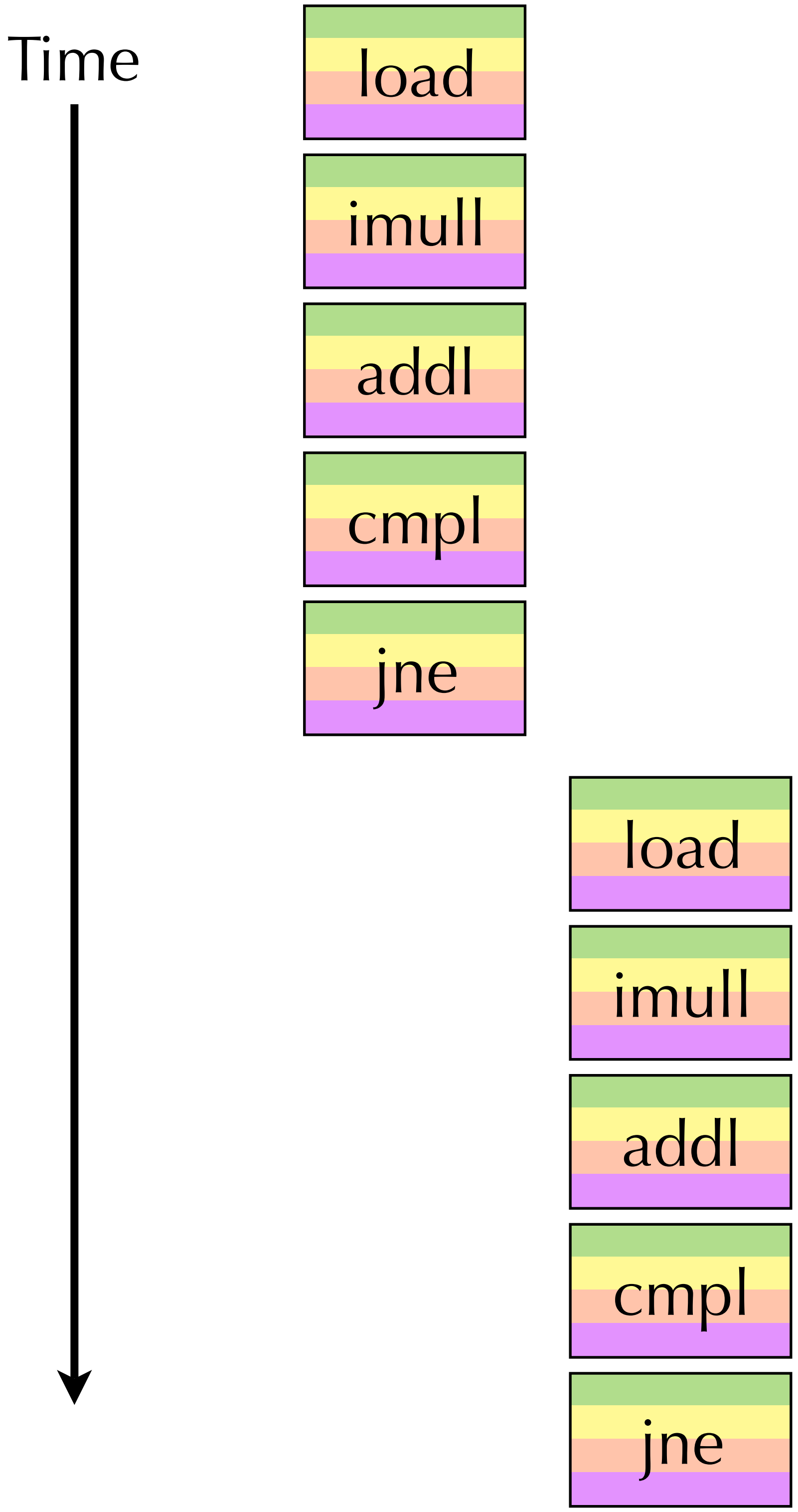
```
int prod_array(int *a, int n) {
  int i, result=1;
  for (i = 0; i < n; i++) {
    result *= a[i];
  }
  return result;
}
```

```
    # Main body of loop
    .L56:
        imull   (%ebx,%edx,4), %eax
        addl    $1, %edx
        cmpl    %edx, %ecx
        jne     .L56
```

- Assume executing an instruction has 4 stages



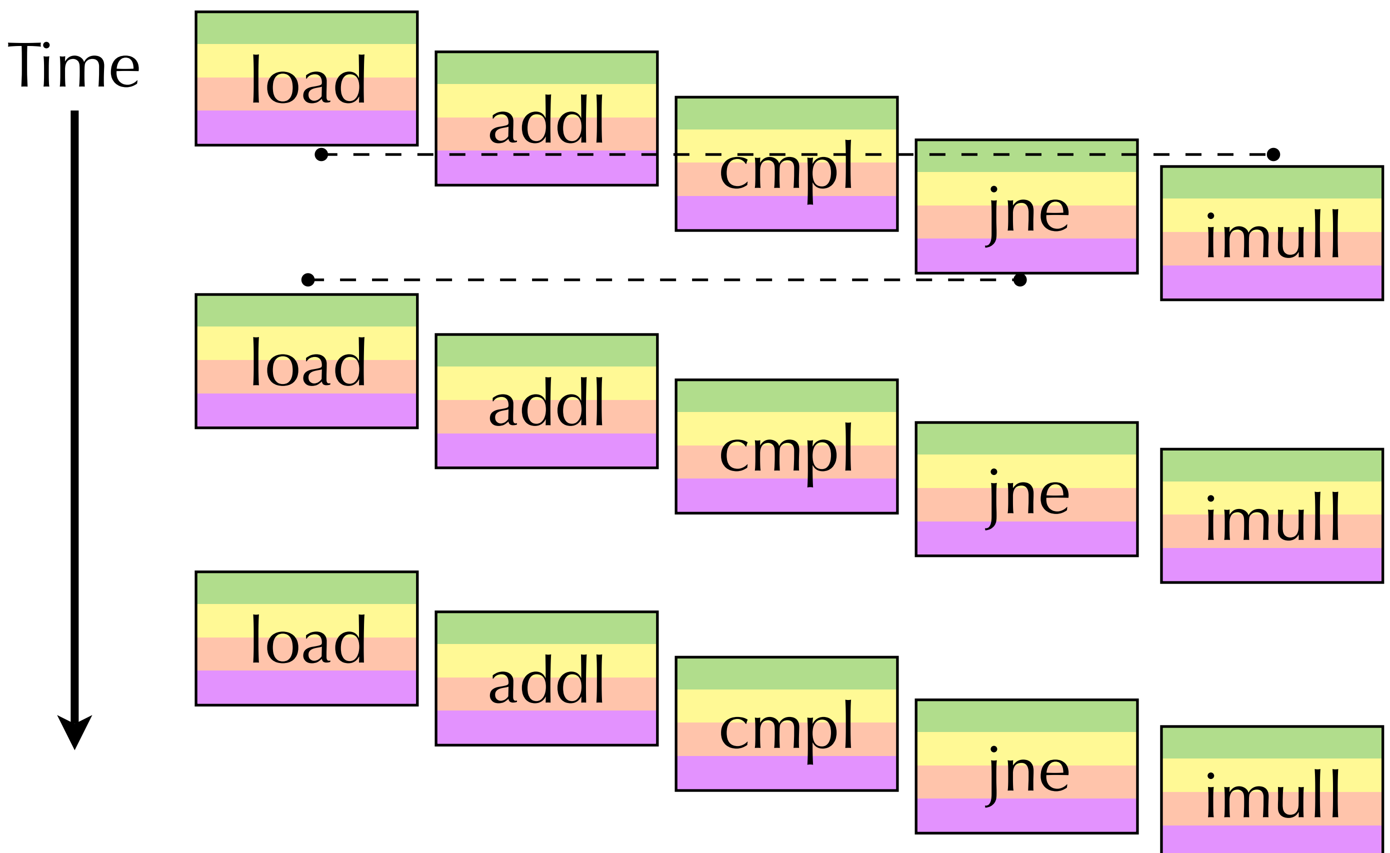
Non-pipelined execution




```
# Main body of loop
.L56:
    imull   (%ebx,%edx,4), %eax
    addl    $1, %edx
    cmpl    %edx, %ecx
    jne     .L56
```

- Each instruction must wait for previous to complete

Pipelined execution

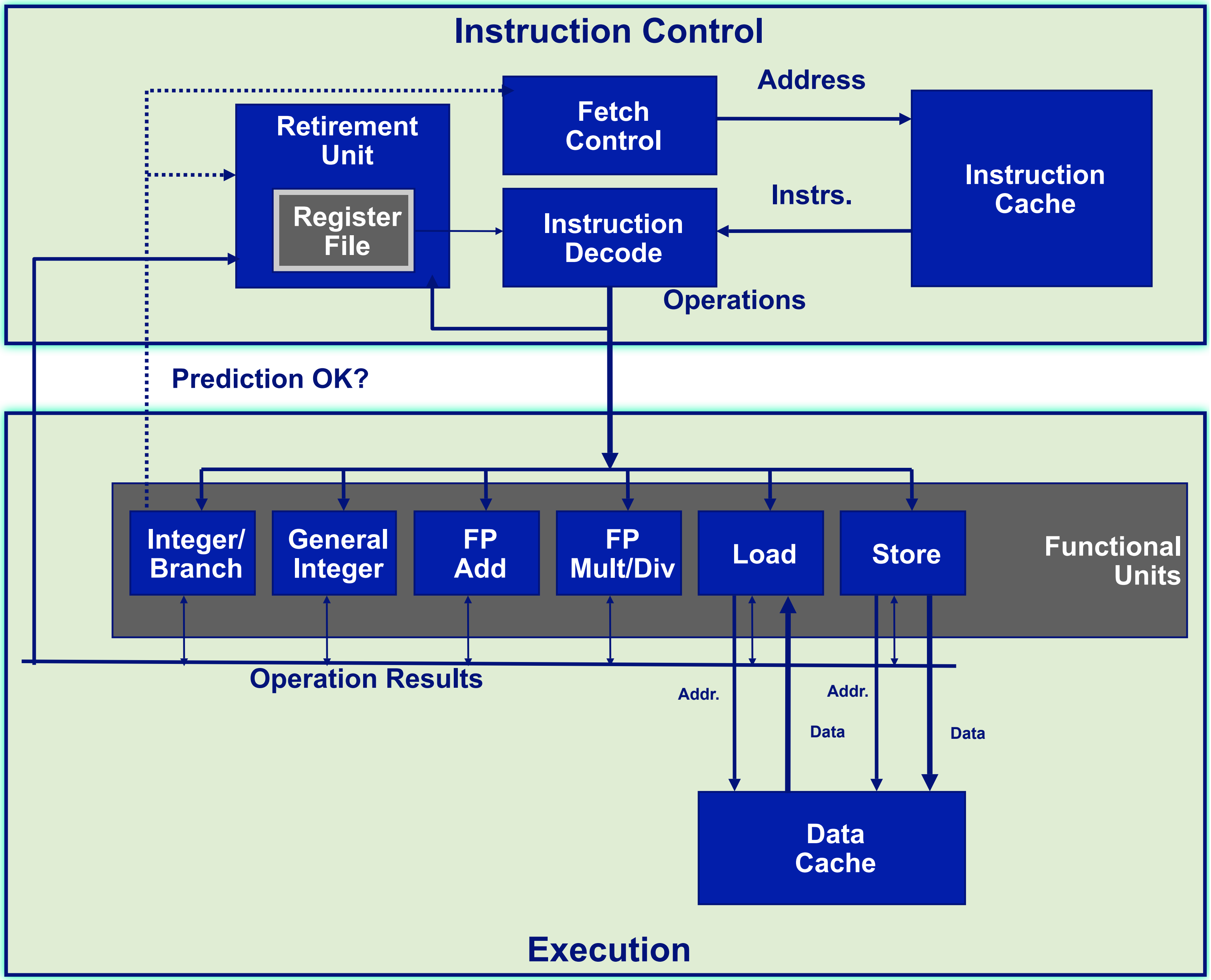



```
# Main body of loop
.L56:
    imull   (%ebx,%edx,4), %eax
    addl    $1, %edx
    cmpl    %edx, %ecx
    jne     .L56
```

- Every stage of pipeline can be processing one instruction
- **Out-of-order** execution
  - `imull` needs result of load, can't start executing `imull` until load finishes
  - But can start executing `addl` while load is still occurring!
  - More efficient to schedule `addl` before `imull`
- Dependencies between instructions can cause "bubbles" in pipeline
- Skipping over many many many details!

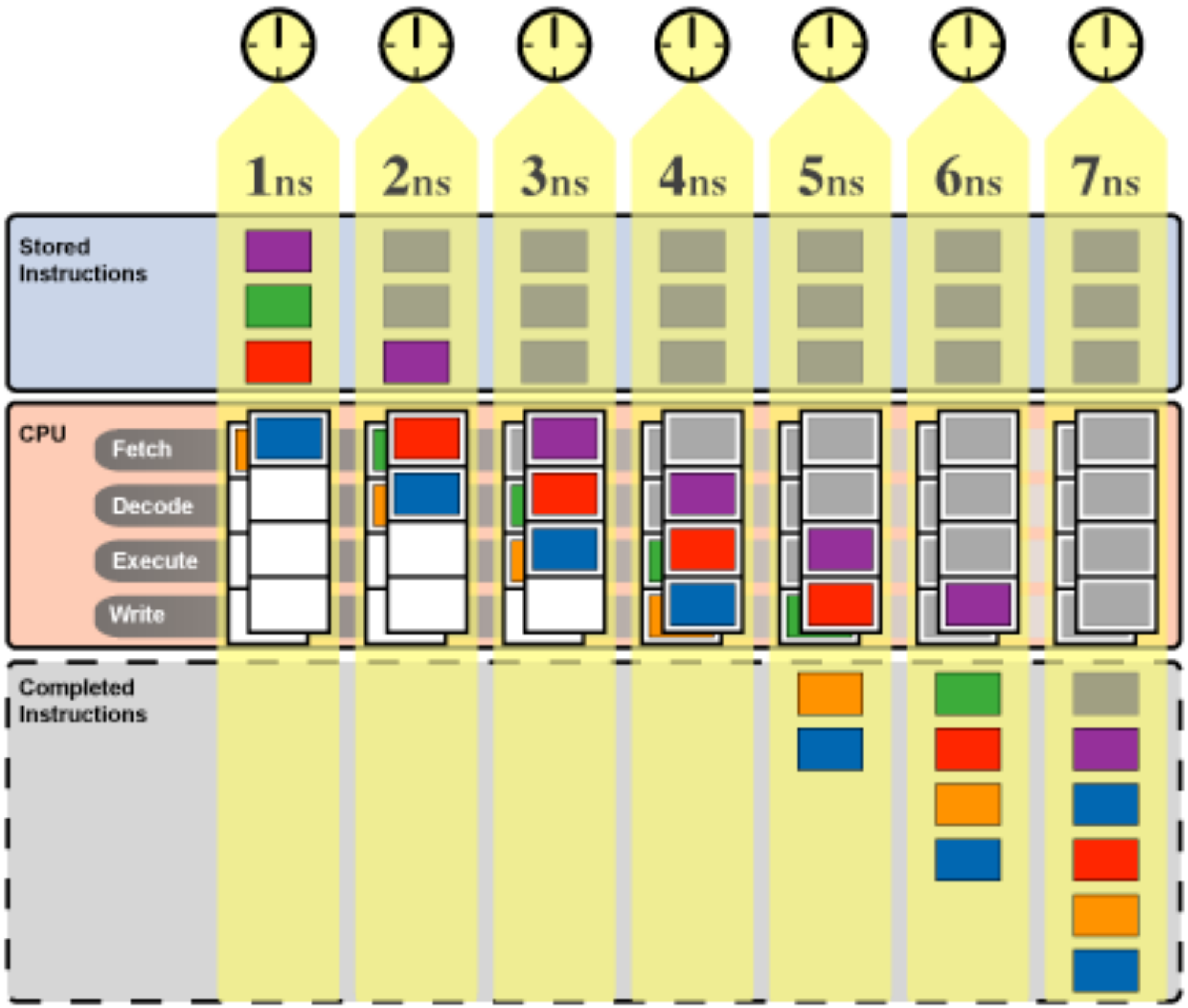
Superscalar processors

- CPU has multiple functional units
  - Each can deal with different kinds of operations
  - Some overlap, e.g., most functional units can do integer arithmetic
- Each functional unit has its own pipeline
  - ⇒ Multiple pipelines executing in parallel



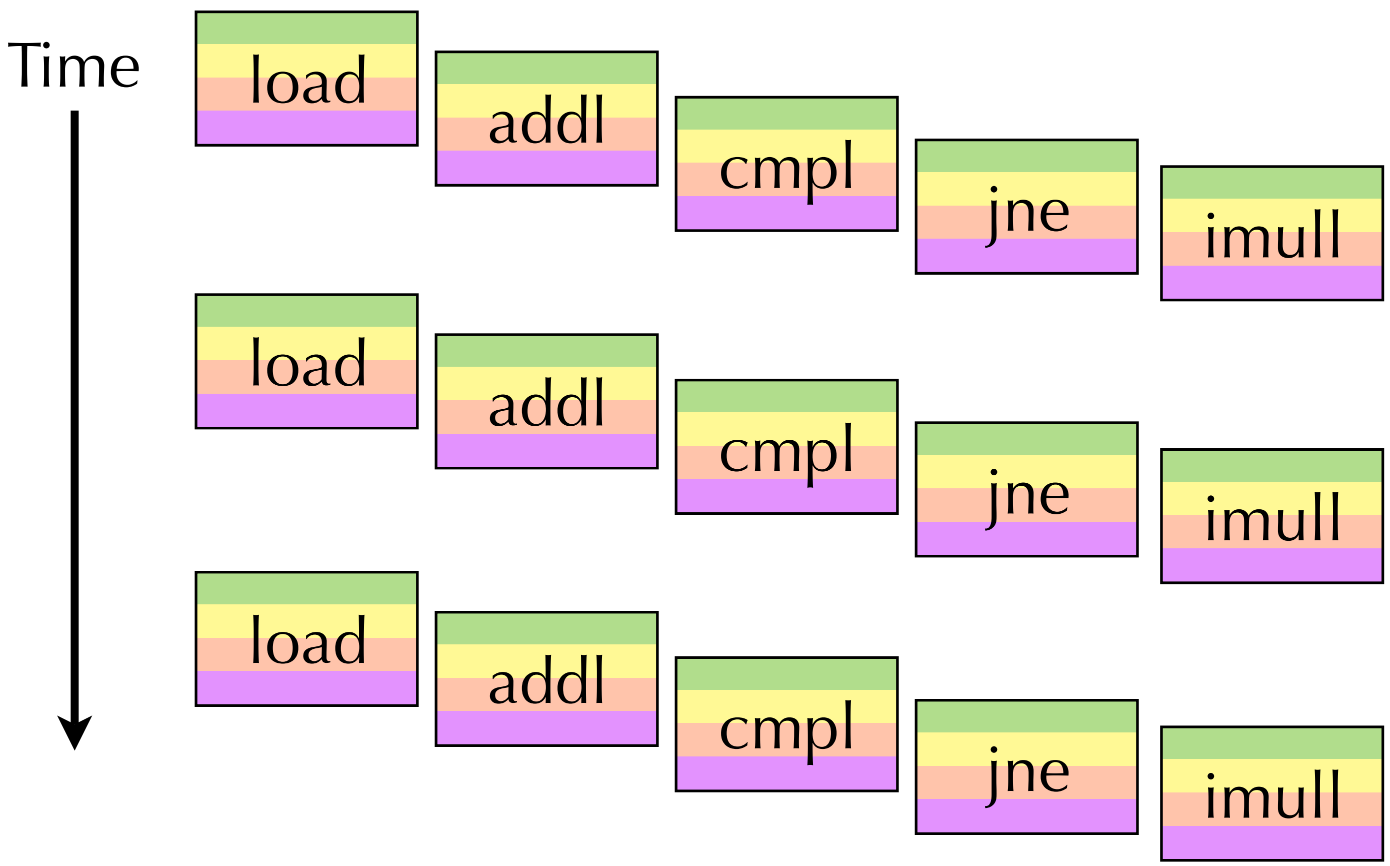
Superscalar processors

- In one cycle can issue different instructions to different functional units
  - Hence in one cycle can complete more than one operation
  - Thus, “superscalar”
- Not the same as multicore!



http://arstechnica.com/old/content/2004/09/pipelining-2.ars/5

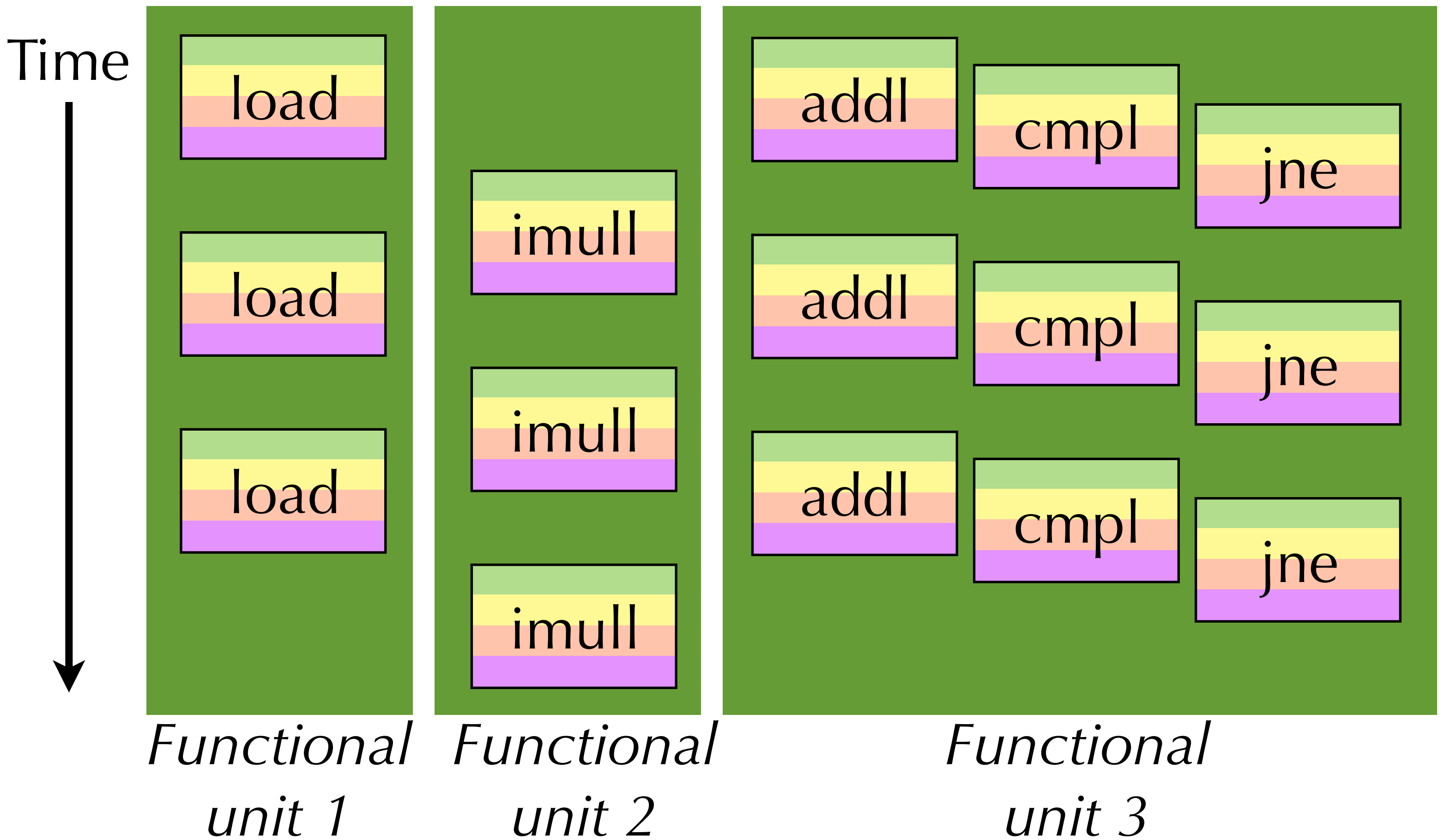
Reminder: Pipelined execution




```
# Main body of loop
.L56:
    imull   (%ebx,%edx,4), %eax
    addl    $1, %edx
    cmpl    %edx, %ecx
    jne     .L56
```

- Each stage of pipeline can be processing at most one instruction
- Different functional units on processor ⇒ Multiple pipelines!
  - Multiple instructions can be issued in one cycle
- (Again, skipping over many details)

Superscalar

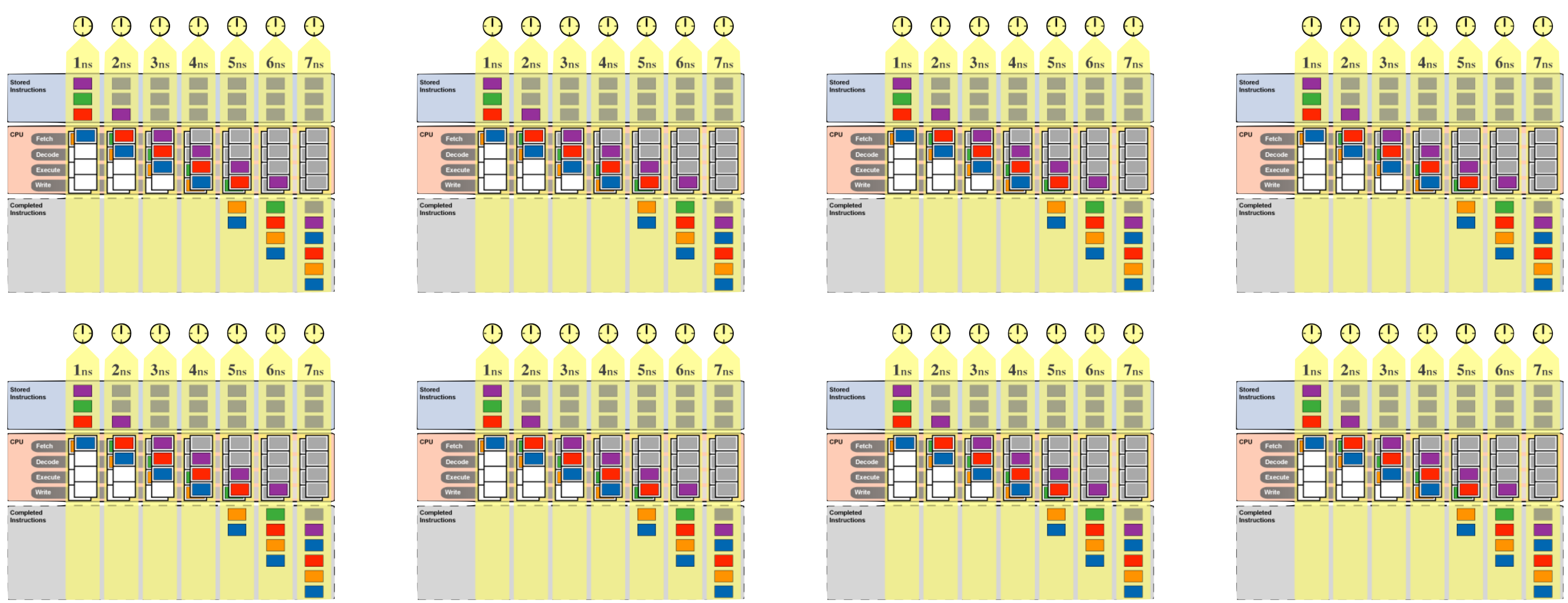



```
# Main body of loop
.L56:
    imull   (%ebx,%edx,4), %eax
    addl    $1, %edx
    cmpl    %edx, %ecx
    jne     .L56
```

- Each stage of pipeline can be processing at most one instruction
- Different functional units on processor ⇒ Multiple pipelines!
  - Multiple instructions can be issued in one cycle
- (Again, skipping over many details)

Multicore processors

- Each chip contains multiple separate processor cores
- Each core can run completely different code
- To take advantage (in a single program) of multiple cores must write *concurrent code*. More on this later in course...



Today

- Program optimization
  - Overview
  - Code motion
  - Strength reduction
  - Common subexpressions
  - Optimization blockers
    - Procedure calls
    - Aliasing
  - Understanding modern processors
  - Loop unrolling
  - Summary

Loop unrolling

- Reduce number of iterations of loop by doing more work each iteration
  - Reduces number of loop index/comparison operations
  - Further transformations can enable additional speedup.

```
int prod_array(int *a, int n) {
  int i, result=1;
  for (i = 0; i < n; i++) {
    result *= a[i];
  }
  return result;
}
```

```
/* Note: assuming n is even!  */
int prod_array2(int *a, int n) {
  int i, tmp1=1, tmp2=1;
  for (i = 0; i < n; i+=2) {
    result *= a[i];
    result *= a[i+1];
  }
  return result;
}
```

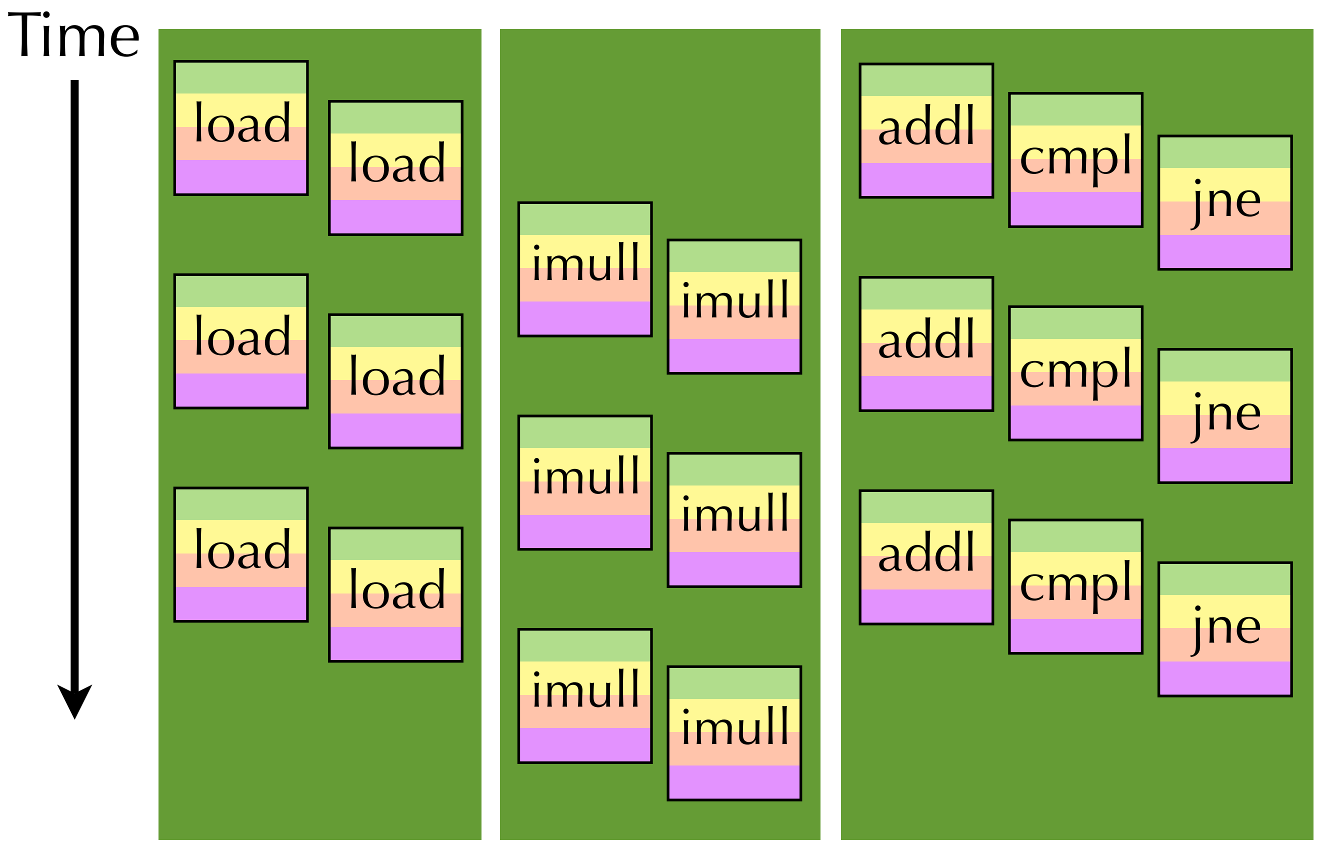
Enhancing parallelism

- In unrolled version, multiplications must occur in sequence
  - Why?
- What if we used two accumulators?
  - No dependency between two multiplications
  - Can be run in parallel

```
/* Note: assuming n is even!  */
int prod_array2(int *a, int n) {
  int i, tmp1=1, tmp2=1;
  for (i = 0; i < n; i+=2) {
    result *= a[i];
    result *= a[i+1];
  }
  return result;
}
```

```
/* Note: assuming n is even!  */
int prod_array2(int *a, int n) {
  int i, tmp1=1, tmp2=1;
  for (i = 0; i < n; i+=2) {
    tmp1 *= a[i];
    tmp2 *= a[i+1];
  }
  return tmp1 * tmp2;
}
```

Visualizing two way unrolling




```
# Unrolled loop
.L77:
    imull   (%ebx,%edx,4), %ecx
    imull   4(%ebx, %edx, 4), %eax
    addl    $2, %edx
    cmpl    %edx, %esi
    jg     .L70
```

- More parallelism for each iteration

Optimization summary

- Write code to help the compiler, and CPU, do their jobs well.
  - Remember: The compiler has to be conservative, but you might know better.
- High-level design
  - Choose appropriate algorithms and data structures
- Basic coding principles
  - Avoid optimization blockers
  - Eliminate unnecessary function calls and memory references
- Low-level optimization
  - Unroll loops to reduce overhead and enable further optimizations
  - Find ways to increase instruction-level parallelism
  - Code motion:
    - Move constant expressions outside of loops
    - Especially in the presence of function calls
  - Strength reduction
    - Use less expensive operations/functions when possible (Though, most compilers will do this for you!)

Caveats

- Does this mean you should write crazy, convoluted, repetitive, but high performing code?
- Probably not.
- Need to balance maintainability/readability with performance
- Always clearly comment when you are doing something funky
    - State your assumptions: someone may change your code later and break it it subtle ways

Caveats

There is no doubt that the grail of efficiency leads to abuse. Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time: ***premature optimization is the root of all evil.***

Yet we should not pass up our opportunities in that critical 3%.

Donald Knuth

Structured Programming with goto Statements
Computing Surveys, Vol 6, No 4, December 1974

How to find the 3%...

- Identifying and eliminating performance bottlenecks
  - Use a program profiler to find out where your program is spending its time
    - e.g., gprof
  - Speed up of program depends on how much you improved performance of component, and how significant component is

Next lecture

- Linking and loading
  - How does the compiler generate a binary?
  - How does the binary get running on the machine?