Announcement

- **Lab2: server mishap (-_-)#**
  - We will round everyone’s score up to the phase that you did
  - Please resubmit your answers *again* by 10pm TONIGHT if you are not on the scoreboard.
- **Next lecture: finish up Program Optimization and Midterm review**
- **A sample midterm will be posted by tonight**
- **Midterm next Wednesday (Oct 16) after the Fall break**
- **Instructor out of town next week (a TA will proctor)**
- **Additional office hours this Friday**

Program Optimization (Cont'd)

B&O Readings: 5

CSE 361: Introduction to Systems Software

Instructor:
I-Ting Angelina Lee

Limitations of Optimizing Compilers

- **Operate under fundamental constraint**
  - Must not change in program behavior; the compiler must be conservative
  - Even under pathological conditions
- **Obvious to programmer ≠ provable to compiler**
  - Behavior can be obfuscated by languages and coding styles
    - e.g., data ranges may be more limited than variable types suggest
- **Most analysis is:**
  - Performed mostly within procedures (whole-program analysis is expensive)
  - Newer GCC performs interprocedural analysis within a single file
  - Based only on *static* information (hard to anticipate run-time inputs)

Today

- Overview
- Machine-Independent Optimizations
  - Code motion/precomputation
  - Strength reduction
  - Common Subexpression Elimination
  - Removing unnecessary procedure calls
- **Optimization Blockers**
  - Procedure calls
  - Memory aliasing
- Exploiting Instruction-Level Parallelism
- Dealing with Conditionals

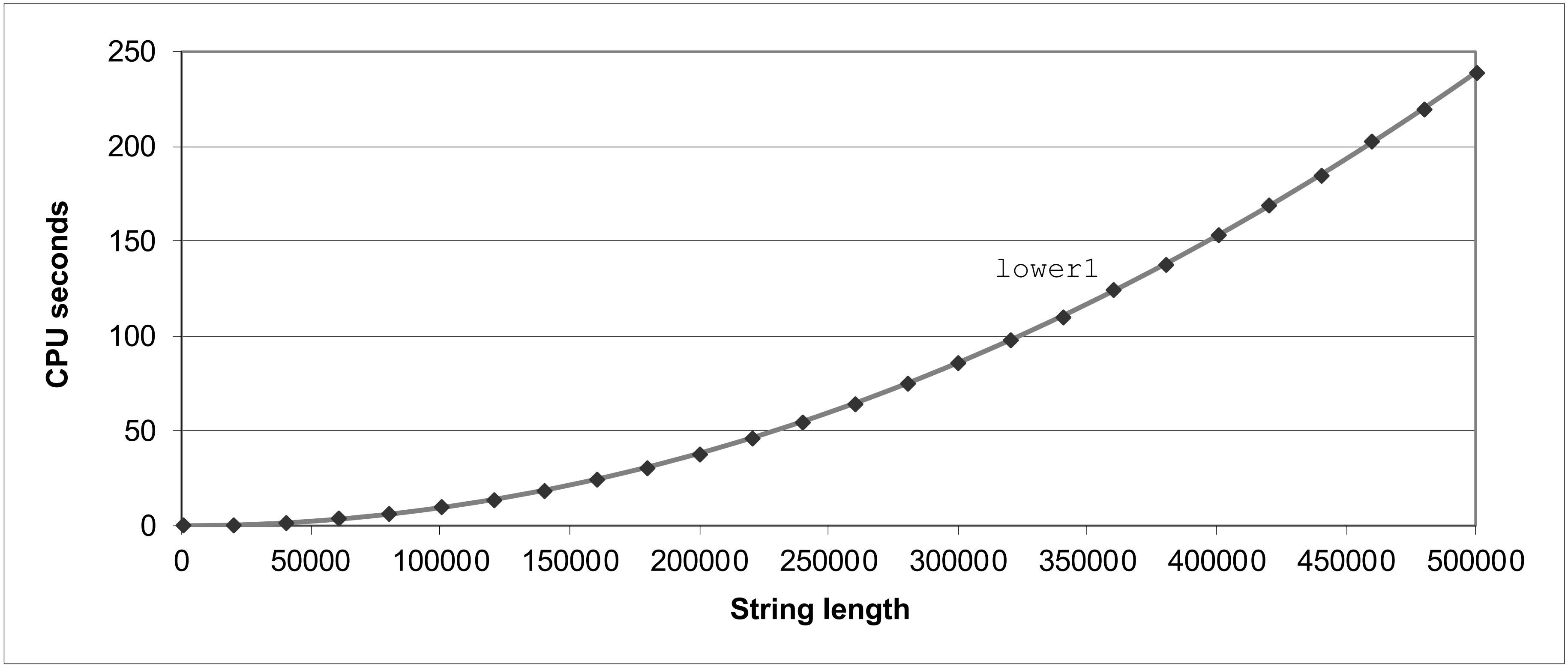
What's Wrong with This Code?

- **Procedure to Convert String to Lower Case**

```
void lower(char *s)
{
  size_t 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 double string length
- Quadratic performance



Convert Loop To Goto Form

```
void lower(char *s)
{
   size_t 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

```
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
  - Each call requires time N
  - Overall $O(N^2)$ performance

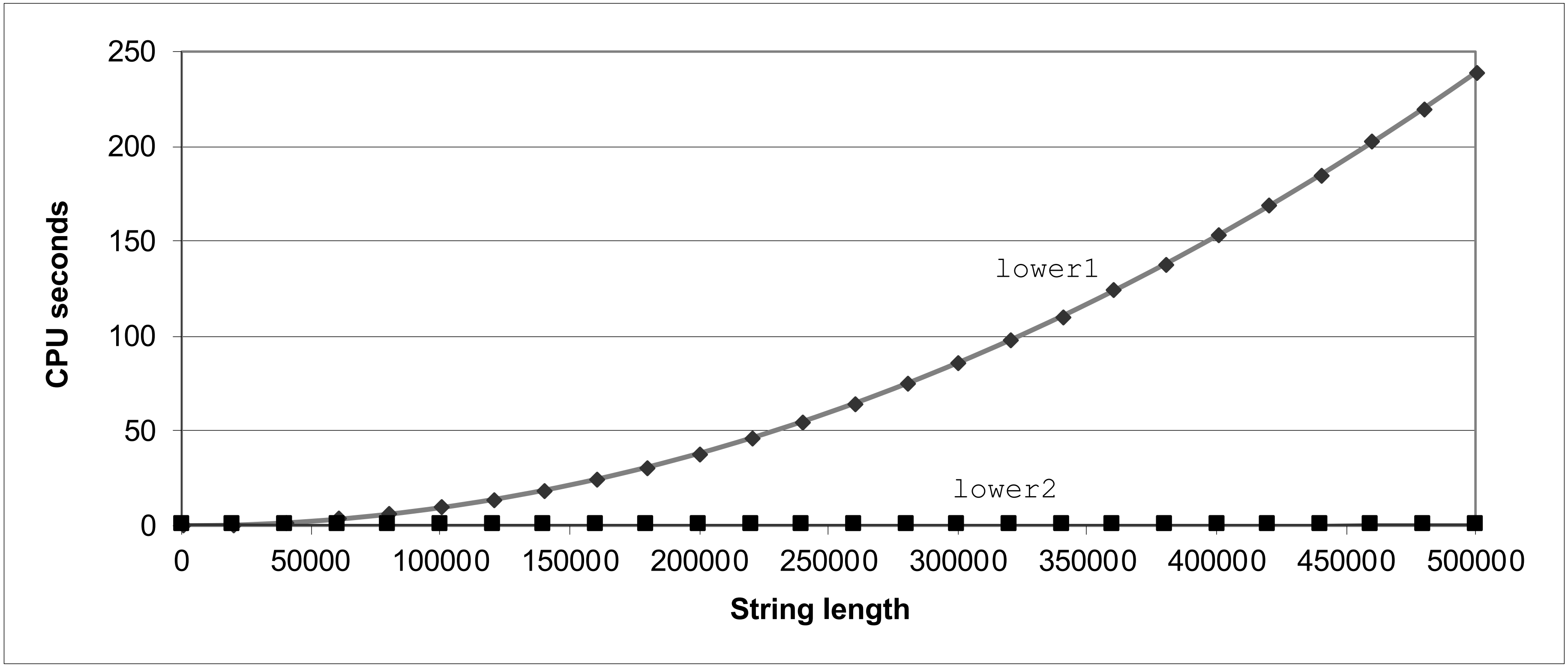
Improving Performance

```
void lower(char *s)
{
  size_t i;
  size_t 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 double string length
- Linear performance of lower2



Optimization Blocker #1: 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 –O1
      - Within single file
  - Do your own code motion

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

Memory Matters

```
/* Sum rows is of n X n matrix a
   and store in vector b  */
void sum_rows1(double *a, double *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];
        }
    }
}
```

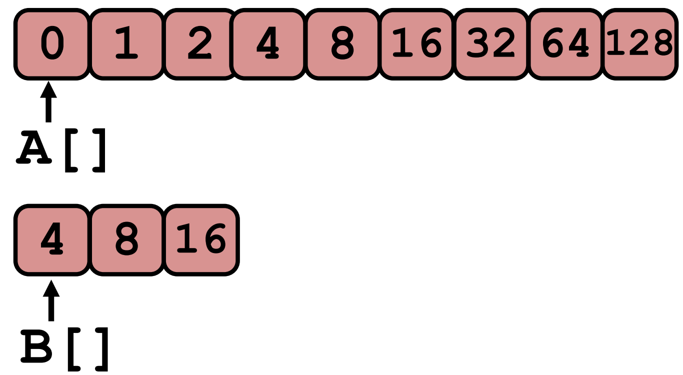
```
# sum_rows1 inner loop
.L4:
        movsd   (%rsi,%rax,8), %xmm0    # FP load b[i]
        addsd   (%rdi), %xmm0           # FP add a[i*n+j]
        movsd   %xmm0, (%rsi,%rax,8)    # FP store b[i]
        addq    $8, %rdi                # advance j
        cmpq    %rcx, %rdi
        jne     .L4
```

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

Scenario: No Memory Aliasing

```
/* Sum rows is of n X n matrix a
   and store in vector b  */
void sum_rows1(double *a, double *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];  //b[i] += a[i][j]
    }
}
```

Initialization:




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

double B[3] = {4,8,16};

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

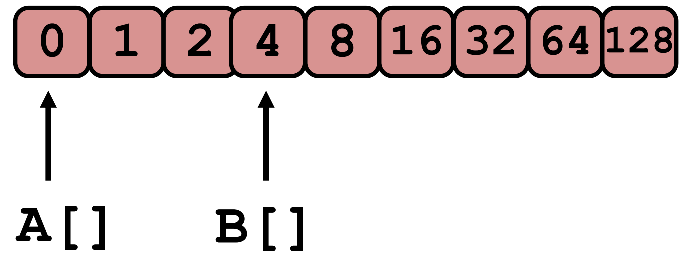
Value of B:

```
init:   [4,8,16]

i = 0:  [0,8,16],[0,8,16],[1,8,16],[3,8,16]

i = 1:  [3,0,16],[3,4,16],[3,12,16],[3,28,16]

i = 2:  [3,28,0],[3,28,32],[3,28,96],[3,28,224]
```

- It doesn't seem to matter whether we update `b[i]` immediately.

Scenario: Memory Aliasing Between A and B

```
/* Sum rows is of n X n matrix a
   and store in vector b  */
void sum_rows1(double *a, double *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];  //b[i] += a[i][j]
    }
}
```




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

double B[3] = A+3;

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

Value of B:

```
init:  [4,8,16]
```

```
i = 0: [0,8,16],[0,8,16],[1,8,16],[3,8,16]
```

```
i = 1: [3,0,16],[3,3,16],[3,6,16],[3,22,16]
```

```
i = 2: [3,22,0],[3,22,32],[3,22,96],[3,22,224]
```

- **`b[i]`** could be the same as **`a[i*n+j]`** if there is aliasing
- As we update **`b[i]`** in inner loop, what you read from **`a[i*n+j]`** changes (such updates change program behavior!).
- Compiler must be conservative and write back the updates immediately.

Tell the Compiler Not to Check Aliasing

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

```
# sum_rows2 inner loop
.L10:
        addsd   (%rdi), %xmm0 # FP load + add
        addq    $8, %rdi
        cmpq    %rax, %rdi
        jne     .L10
```

- Introduce a local variable **`val`**
- This tells the compiler that no need to write back intermediate update.
- Note that this would change the results if A and B alias.

Optimization Blocker #2: 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

Today

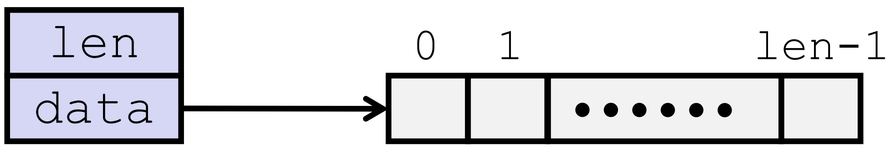
- Overview
- Machine-Independent Optimizations
  - Code motion/precomputation
  - Strength reduction
  - Common Subexpression Elimination
  - Removing unnecessary procedure calls
- Optimization Blockers
  - Procedure calls
  - Memory aliasing
- **Exploiting Instruction-Level Parallelism**
- Dealing with Conditionals

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 yield dramatic performance improvement**
  - Compilers often cannot make these transformations
  - Lack of associativity and distributivity in floating-point arithmetic

Benchmark Example: Data Type for Vectors

```
/* data structure for vectors */
typedef struct{
    size_t len;
    data_t *data;
} vec;
```



- **Data Types**
  - Use different declarations for `data_t`
  - `int`
  - `long`
  - `float`
  - `double`

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

Benchmark Computation

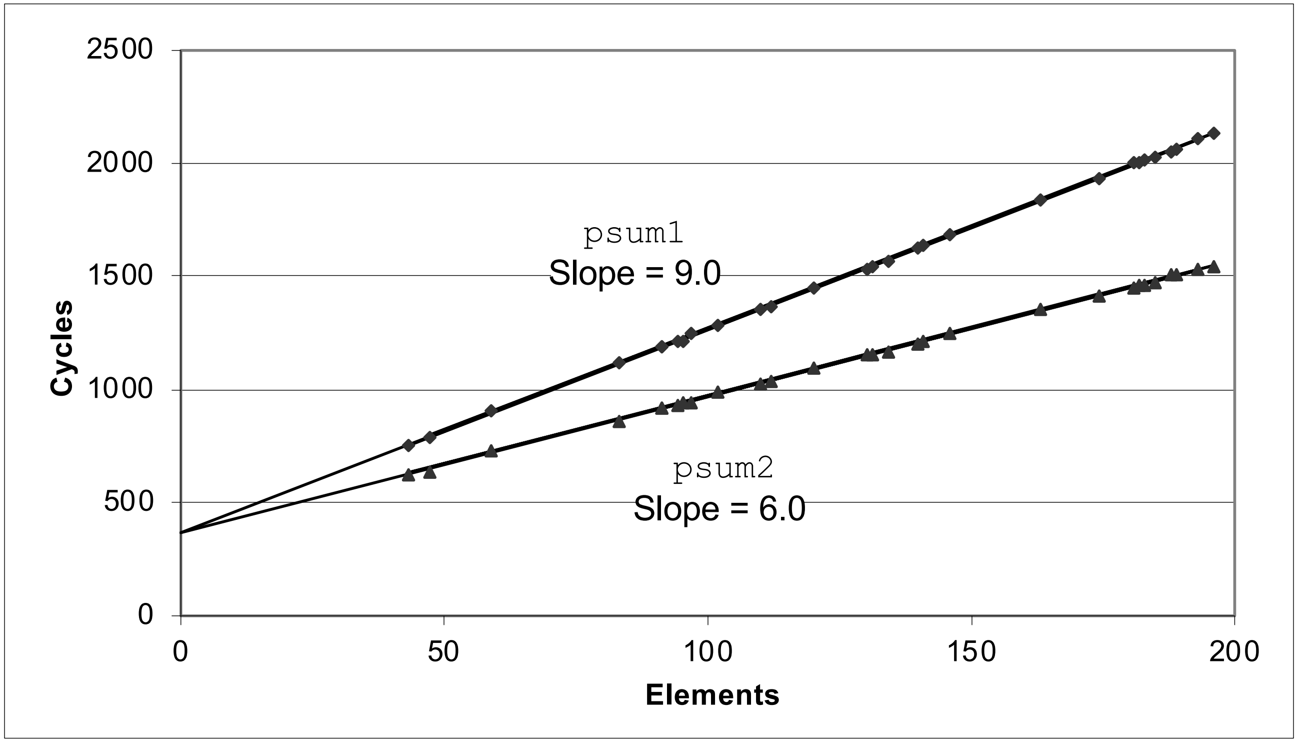
```
void combine1(vec_ptr v, data_t *dest)
{
    long int i;
    *dest = IDENT;
    for (i = 0; i < vec_length(v); i++) {
      data_t val;
      get_vec_element(v, i, &val);
      *dest = *dest OP val;
    }
}
```

Compute sum or product of vector elements

- **Data Types**
  - Use different declarations for `data_t`
  - `int`
  - `long`
  - `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**
- **In our case: CPE = cycles per OP**
- **T = CPE*n + Overhead**
  - CPE is slope of line



Benchmark Performance

```
void combine1(vec_ptr 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	22.68	20.02	19.98	20.18
Combine1 –O1	10.12	10.12	10.17	11.14
Combine1 –O3	4.5	4.5	6	7.8

Results in CPE (cycles per element)

Basic Optimizations

```
void combine4(vec_ptr v, data_t *dest)
{
  long i;
  long 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;
}
```

- Move vec_length out of loop
- Avoid bounds check on each cycle
- Accumulate in temporary

Effect of Basic Optimizations

```
void combine4(vec_ptr v, data_t *dest)
{
  long i;
  long 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 –O3	4.5	4.5	6	7.8
Combine4	1.27	3.01	3.01	5.01

- Eliminates sources of overhead in loop

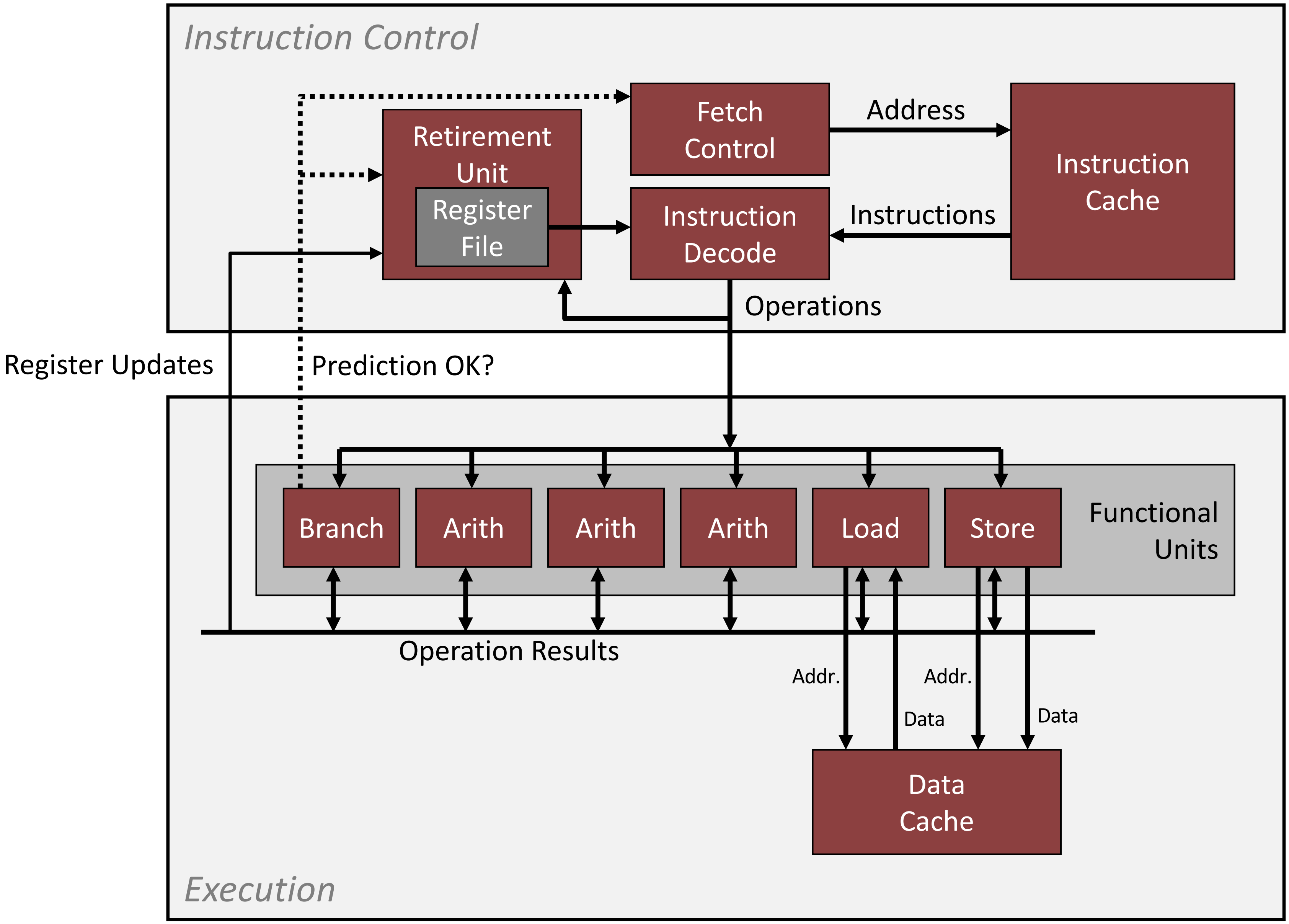
Superscalar Processor

- **Definition:** A superscalar processor can issue and execute *multiple instructions in one cycle*. The instructions are retrieved from a sequential instruction stream and are usually scheduled dynamically.

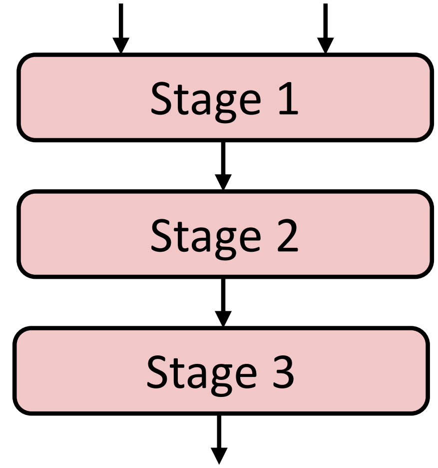
- Benefit: without programming effort, superscalar processor can take advantage of the *instruction level parallelism* that most programs have

- Most modern CPUs are superscalar.
- Intel: since Pentium (1993)

Modern CPU Design



Pipelined Functional Units




```
long mult_eg(long a, long b, long c) {
    long p1 = a*b;
    long p2 = a*c;
    long p3 = p1 * p2;
    return p3;
}
```

	Time						
	1	2	3	4	5	6	7
Stage 1	a*b	a*c			p1*p2		
Stage 2		a*b	a*c			p1*p2	
Stage 3			a*b	a*c			p1*p2

- Divide computation into stages
- Pass partial computations from stage to stage
- Stage i can start on new computation once values passed to i+1
- E.g., complete 3 multiplications in 7 cycles, even though each requires 3 cycles (much better throughput)

Haswell CPU

- **Multiple instructions can execute in parallel**
  - 2 load, with address computation
  - 1 store, with address computation
  - 4 integer
  - 2 FP multiply
  - 1 FP add
  - 1 FP divide
- **Some instructions take > 1 cycle, but can be pipelined**

Instruction	*Latency*	*Cycles/Issue*
Load / Store	4	1
Integer Multiply	3	1
Integer/Long Divide	**3-30**	**3-30**
Single/Double FP Multiply	5	1
Single/Double FP Add	3	1
Single/Double FP Divide	**3-15**	**3-15**

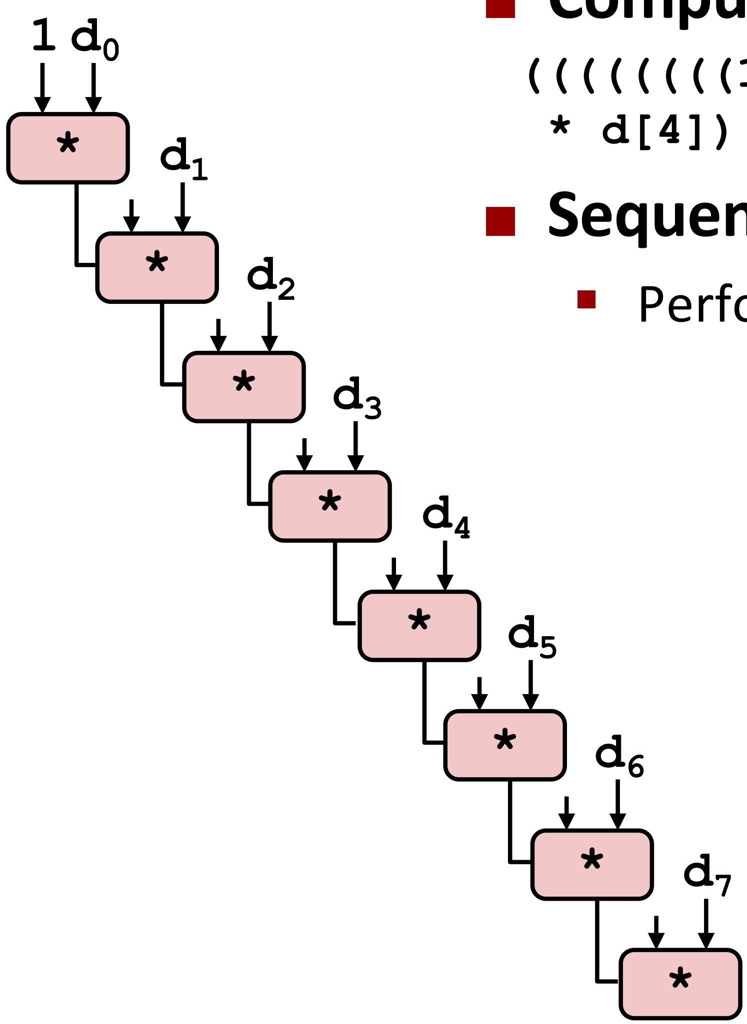
x86-64 Compilation of Combine4

- **Inner Loop (Case: Integer Multiply)**

```
.L519:                            # Loop:
  imull  (%rax,%rdx,4), %ecx      # t = t * d[i]
  addq   $1, %rdx                 # i++
  cmpq   %rdx, %rbp               # Compare length:i
  jg     .L519                    # If >, goto Loop
```

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	1.27	3.01	3.01	5.01
Latency Bound	1.00	3.00	3.00	5.00

Combine4 = Serial Computation (OP = *)



- **Computation (length=8)**

```
((((((((1 * d[0]) * d[1]) * d[2]) * d[3])
 * d[4]) * d[5]) * d[6]) * d[7])
```

- **Sequential dependence**
  - Performance: determined by latency of OP

Loop Unrolling (2x1)

```
void unroll2a_combine(vec_ptr v, data_t *dest)
{
    long length = vec_length(v);
    long limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x = IDENT;
    long i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
        x = (x OP d[i]) OP d[i+1];
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        x = x OP d[i];
    }
    *dest = x;
}
```

- Perform 2x more useful work per iteration

Effect of Loop Unrolling

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	1.27	3.01	3.01	5.01
Unroll 2x1	1.01	3.01	3.01	5.01
Latency Bound	1.00	3.00	3.00	5.00

- **Helps integer add**
  - Achieves latency bound
  - More work per index-increment

```
x = (x OP d[i]) OP d[i+1];
```

- **Others don't improve. *Why?***
  - Still sequential dependency across loop iterations

Loop Unrolling with Reassociation (2x1a)

```
void unroll2aa_combine(vec_ptr v, data_t *dest)
{
    long length = vec_length(v);
    long limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x = IDENT;
    long i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
        x = x OP (d[i] OP d[i+1]);
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        x = x OP d[i];
    }
    *dest = x;
}
```

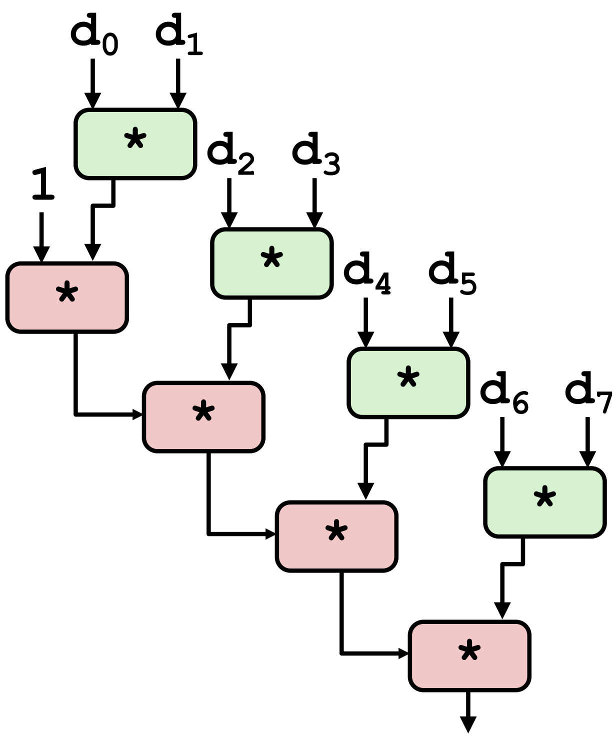
Compare to before

```
x = (x OP d[i]) OP d[i+1];
```

- **Can this change the result of the computation?**
- **Yes, for FP. *Why?***

Reassociated Computation

```
x = x OP (d[i] OP d[i+1]);
```



- **What changed:**
  - Ops in the next iteration can be started early (no dependency)
- **Overall Performance**
  - N elements, D cycles latency/op
  - (N/2+1)*D cycles:
    CPE = D/2