Chapter 5: Optimizing Program Performance

CSCI3240: Lecture 12 and 13

Dr. Arpan Man Sainju

Middle Tennessee State University

Today

- **Overview**
- **Generally Useful Optimizations**
  - Code motion/precomputation
  - Strength reduction
  - Sharing of common subexpressions
  - Removing unnecessary procedure calls
- **Optimization Blockers**
  - Procedure calls
  - Memory aliasing

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 modern processors + memory systems operate
  - 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
  - Newer versions of GCC do inter-procedural analysis within individual files
    - But, not between code in different files
- **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**

- **A. Code Motion**
- **B. Reduction in Strength**
- **C. Share Common Subexpression**

A. Code Motion

- **A. Code Motion**
  - Avoid repeated computation
    - If it will always produce same result
    - Especially moving code out of loop

```
void set_row(double *a, double *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];
```

Compiler-Generated Code Motion (-O1)

```
void set_row(double *a, double *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;
    double *rowp = a+ni;
    for (j = 0; j < n; j++)
     *rowp++ = b[j];
```

```
set_row:
    testq    %rcx, %rcx        # Test n
    jle  .L1            # If 0, goto done
    imulq    %rcx, %rdx        # ni = n*i
    leaq (%rdi,%rdx,8), %rdx   # rowp = A + ni*8
    movl $0, %eax                    # j = 0
.L3:                          # loop:
    movsd    (%rsi,%rax,8), %xmm0      # t = b[j]
    movsd    %xmm0, (%rdx,%rax,8)      # M[A+ni*8 + j*8] = t
    addq $1, %rax             # j++
    cmpq %rcx, %rax           # j:n
    jne  .L3            # if !=, goto loop
.L1:                          # done:
    rep ; ret
```

B. Reduction in Strength

- Replace costly operation with simpler one
- Shift, add instead of multiply or divide

  ```
  16*x  -->   x << 4
  ```

  - Utility machine dependent
  - Depends on cost of multiply or divide instruction
    - On Intel Nehalem, integer multiply requires 3 CPU cycles
- Recognize the sequence of products

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

C. Share Common Subexpressions

- Reuse portions of expressions
- GCC will do this with –O1

```
/* 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;
```

3 multiplications: i*n, (i–1)*n, (i+1)*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
```

```
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;
```

1 multiplication: i*n

```
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
```

Here, n refers to number of columns, i and j indicates row index and column index respectively.

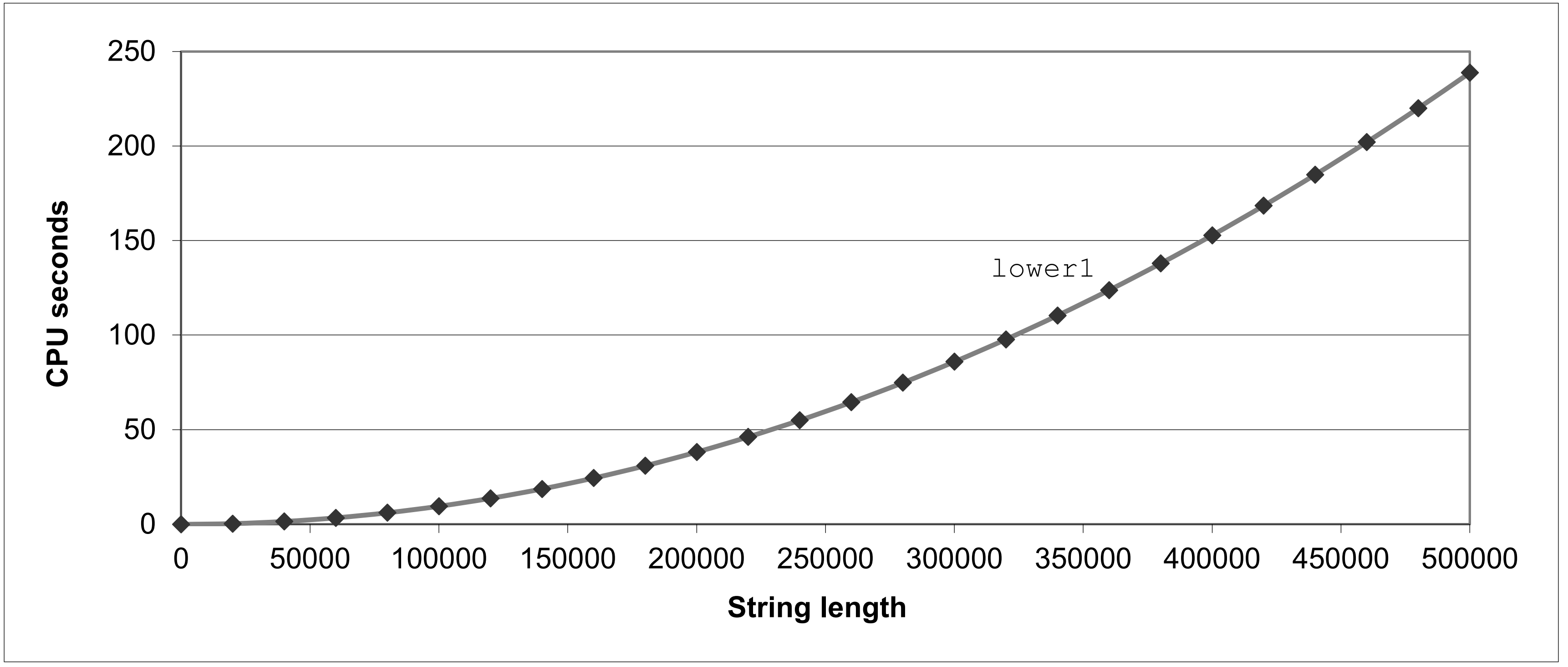
Optimization Blocker #1: Procedure Calls

- **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

```
/* My version of 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
  - Require times N, N-1, N-2, …, 1
  - Overall $O(N^2)$ performance

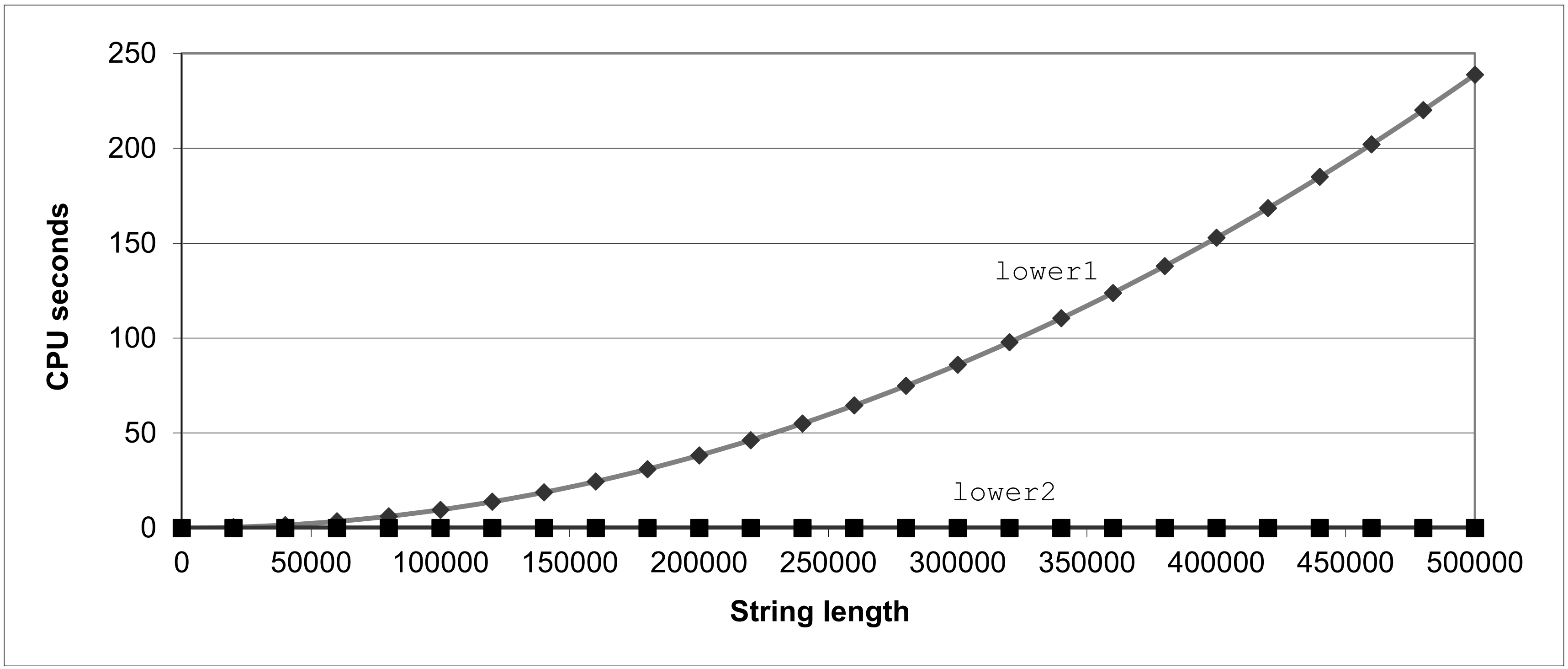
Improving Performance

```
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 (but compiler does not do it for you)

Lower Case Conversion Performance

- Time doubles when double string length
- 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 –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
        addsd   (%rdi), %xmm0         # FP add
        movsd   %xmm0, (%rsi,%rax,8)  # FP store
        addq    $8, %rdi
        cmpq    %rcx, %rdi
        jne     .L4
```

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

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];
    }
}
```

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

double B[3] = A+3;  //B[3] is alias to A+3

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

Value of `B`:

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

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

```
i = 1: [3, 28, 16]
```

```
i = 2: [3, 28, 224]
```

- Code updates `b[i]` on every iteration
- Must consider possibility that these updates will affect program behavior

Removing 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
```

- No need to store intermediate results

Optimization Blocker: Memory Aliasing

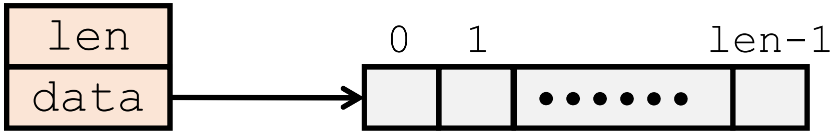
- **Aliasing**
  - Two different memory references specify single location
  - Easy to have happen in C
    - Since allowed to do address arithmetic
    - Direct access to storage structures
  - Get in habit of introducing local variables
    - Accumulating within loops
    - Your way of telling compiler not to check for aliasing

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

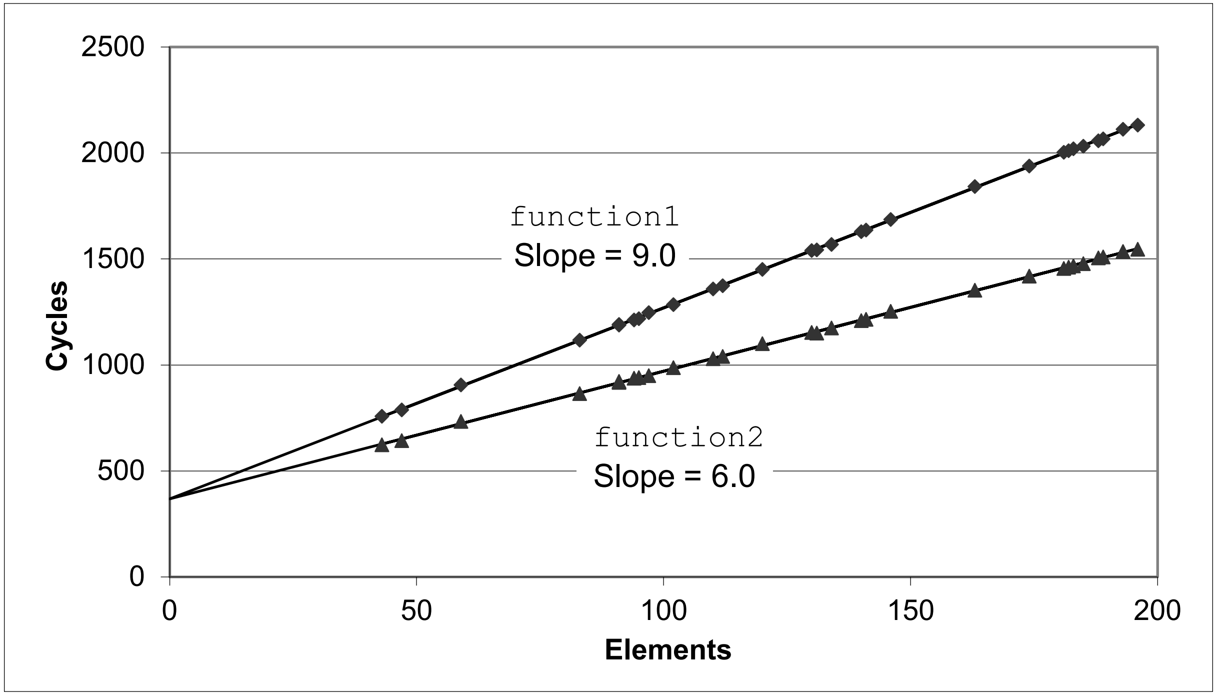
```
void combine1(vec_ptr v, data_t *dest)
{
    long int i;
    *dest = IDENT;
    for (i = 0; i < vec_length(v); i++) {
      data_t val;
      get_vec_element(v, i, &val);
      *dest = *dest OP val;
    }
}
```

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 the performance of a 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

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;
}
```

```
/* Return length of vector */
size_t vec_length(vec *v)
{
    return v->len;
}

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

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

```
/* 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;
}
```

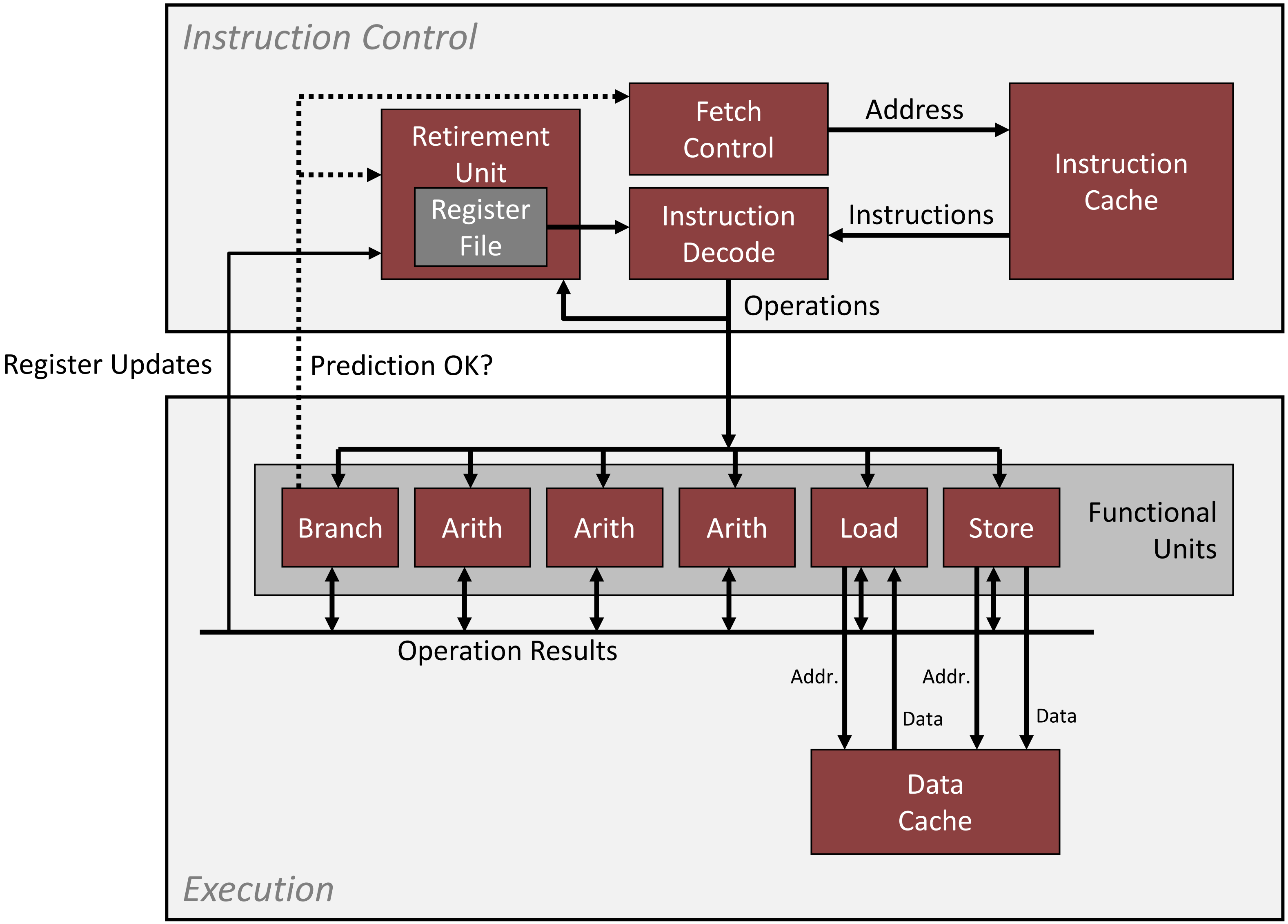
Effect of Basic Optimizations

```
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 –O1	10.12	10.12	10.17	11.14
Combine4	1.27	3.01	3.01	5.01

- Eliminates sources of overhead in loop

Modern CPU Design



Superscalar Processor

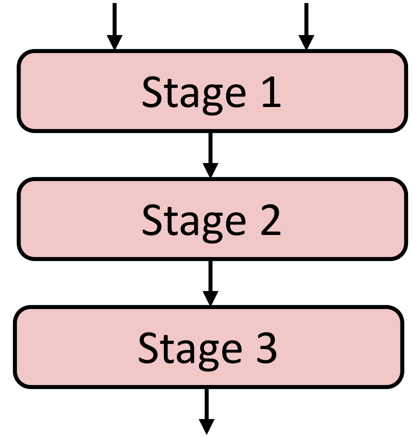
- **Definition:** A superscalar processor can issue and execute ***multiple instructions in one cycle***. The instructions are retrieved from a sequential instruction stream and are usually scheduled dynamically.

- **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)**

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

Haswell CPU

- 8 Total Functional Units

- **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

Latency refers to total number of clock cycles required to perform the operation.

Issue time indicates the minimum number of cycles between two independent operation of same type.

- **Some instructions take > 1 cycle, but can be pipelined**

Instruction	***Latency***	***Cycles/Issue***
Load / Store	4	1
Integer Addition	1	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

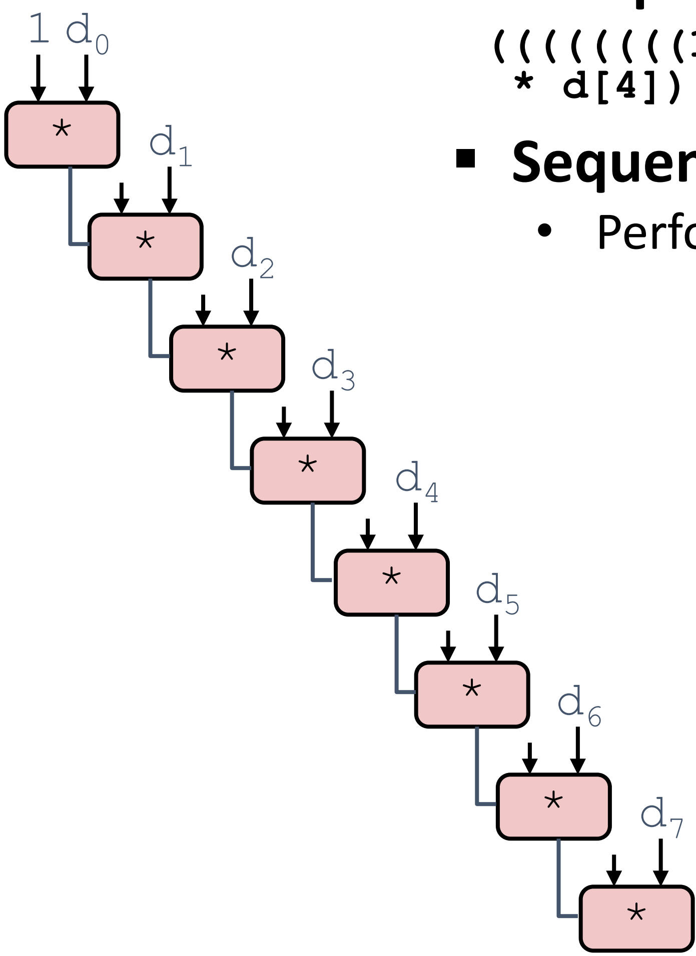
- **Inner Loop (Case: Integer Multiply)**

```
.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

- Here, we have series of multiplication of integers. But we must wait for the imull operation (3 clock cycle bound) to complete before performing the next imull instruction.
- The measurement here corresponds to the latency bound of the machine.

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

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

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

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?***

Effect of Reassociation

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
Unroll 2x1a	1.01	1.51	1.51	2.51
Latency Bound	1.00	3.00	3.00	5.00
Throughput Bound	0.50	1.00	1.00	0.50

2 func. units for FP *
2 func. units for load

4 func. units for int +
2 func. units for load

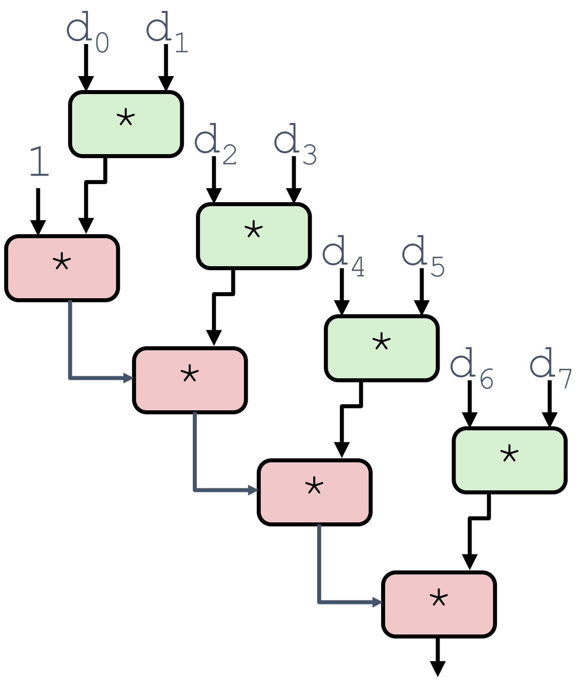
- **Nearly 2x speedup for Int *, FP +, FP ***
  - Reason: Breaks sequential dependency

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

  - Why is that? (next slide)

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

Loop Unrolling with Separate Accumulators (2x2)

```
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 x0 = IDENT;
    data_t x1 = IDENT;
    long i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
       x0 = x0 OP d[i];
       x1 = x1 OP d[i+1];
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        x0 = x0 OP d[i];
    }
    *dest = x0 OP x1;
}
```

- **Different form of reassociation**

Effect of Separate Accumulators

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
Unroll 2x1a	1.01	1.51	1.51	2.51
Unroll 2x2	0.81	1.51	1.51	2.51
Latency Bound	1.00	3.00	3.00	5.00
Throughput Bound	0.50	1.00	1.00	0.50

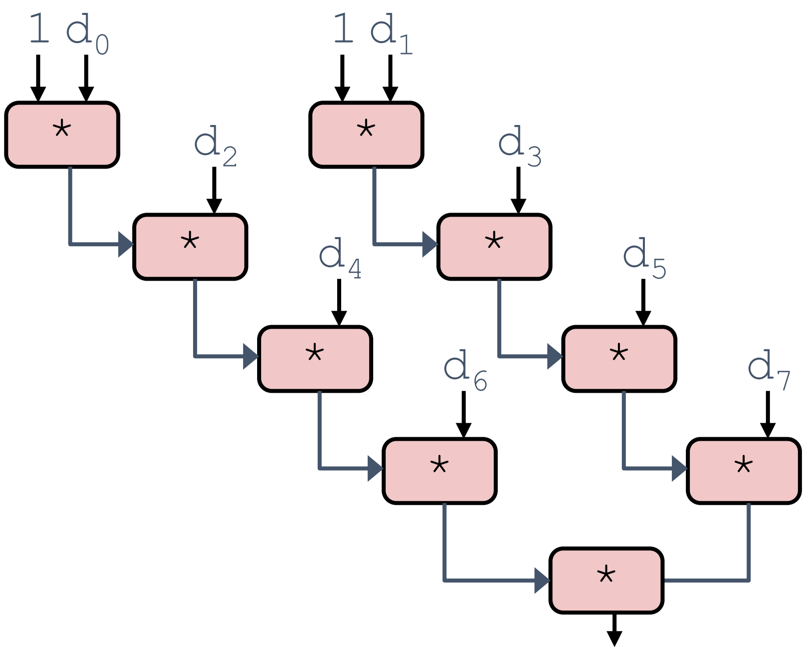
- Int + makes use of two load units

```
x0 = x0 OP d[i];
x1 = x1 OP d[i+1];
```

- 2x speedup (over unroll2) for Int *, FP +, FP *

Separate Accumulators

```
x0 = x0 OP d[i];
x1 = x1 OP d[i+1];
```



- **What changed:**
  - Two independent “streams” of operations
- **Overall Performance**
  - N elements, D cycles latency/op
  - Should be (N/2+1)*D cycles: **CPE = D/2**
  - CPE matches prediction!

What Now?

Unrolling & Accumulating

- **Idea**
  - Can unroll to any degree L
  - Can accumulate K results in parallel
  - L must be multiple of K

- **Limitations**
  - Diminishing returns
    - Cannot go beyond throughput limitations of execution units
  - Large overhead for short lengths
    - Finish off iterations sequentially

Unrolling & Accumulating: Double *

- **Case**
  - Intel Haswell
  - Double FP Multiplication
  - Latency bound: 5.00. Throughput bound: 0.50

FP *	Unrolling Factor L							
K	1	2	3	4	6	8	10	12
1	5.01	5.01	5.01	5.01	5.01	5.01	5.01	
2		2.51		2.51		2.51		
3			1.67					
4				1.25		1.26		
6					0.84			0.88
8						0.63		
10							0.51	
12								0.52

Accumulators

Unrolling & Accumulating: Int +

- **Case**
  - Intel Haswell
  - Integer addition
  - Latency bound: 1.00. Throughput bound: 1.00

Int +	Unrolling Factor L							
K	1	2	3	4	6	8	10	12
1	1.27	1.01	1.01	1.01	1.01	1.01	1.01	
2		0.81		0.69		0.54		
3			0.74					
4				0.69		1.24		
6					0.56			0.56
8						0.54		
10							0.54	
12								0.56

Accumulators

Achievable Performance

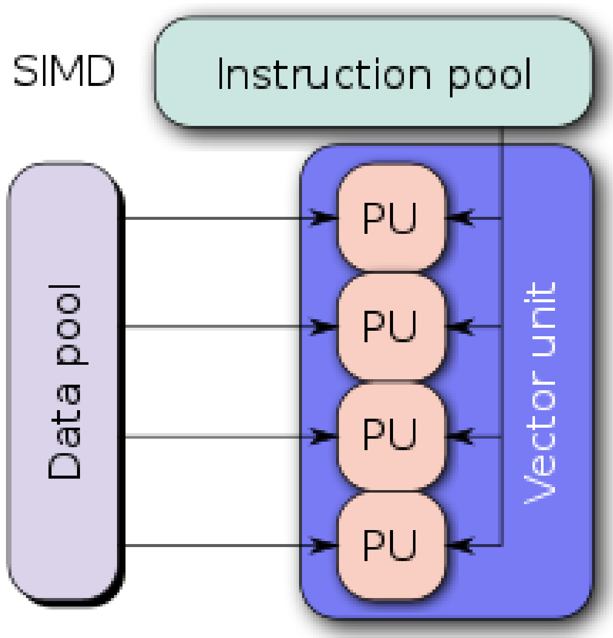
Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Best	0.54	1.01	1.01	0.52
Latency Bound	1.00	3.00	3.00	5.00
Throughput Bound	0.50	1.00	1.00	0.50

- **Limited only by throughput of functional units**
- **Up to 42X improvement over original, unoptimized code**

Floating Point Registers

SIMD

- **SMID refers to "Single Instruction Multiple Data"**



Single Instruction Multiple Data

PU: processing unit

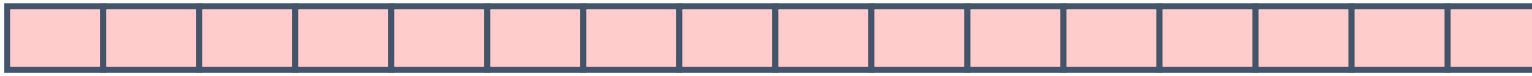
SMID Instruction Set Extensions

- **MMX (1996)**
- **3DNow! (1988)**
- **SSE(1998) [Streaming SMID Extension]**
- **SSE2(2001)**
- **SSE3(2004)**
  - 16 32-bytes registers
- **AVX(2008) [Advance Vector Extension]**
- **AVX2(2013)**
- **AVX-512(2015)**
  - 16 64-bytes registers
- **AMX(2022) [Advance Matrix Extension]**
  - designed to work on matrices to accelerate artificial intelligence (AI) / machine learning (ML) -related workloads.

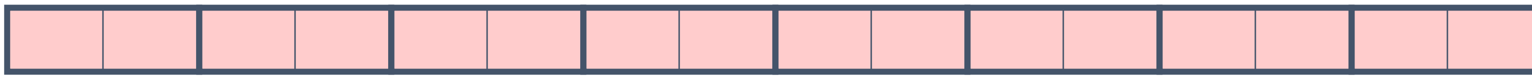
Programming with SSE3

XMM Registers

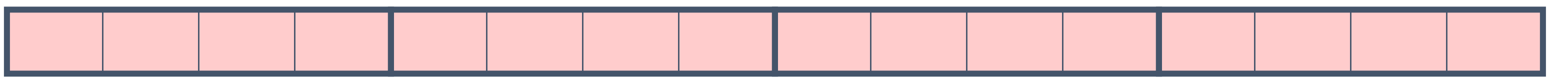
- 16 total, each 16 bytes
- 16 single-byte integers



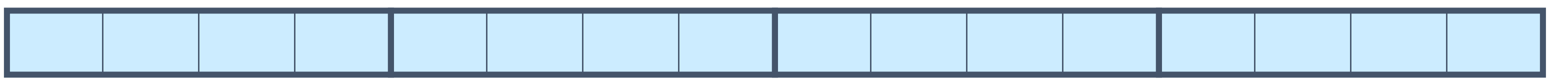
- 8 16-bit integers



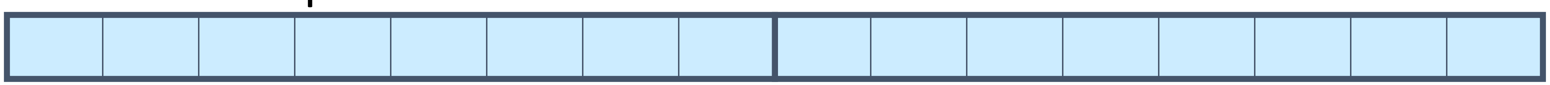
- 4 32-bit integers



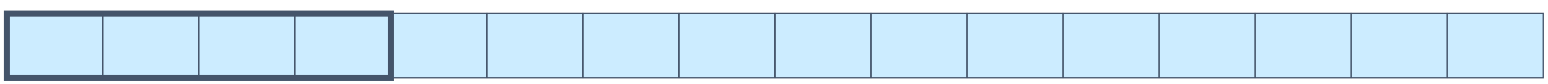
- 4 single-precision floats



- 2 double-precision floats



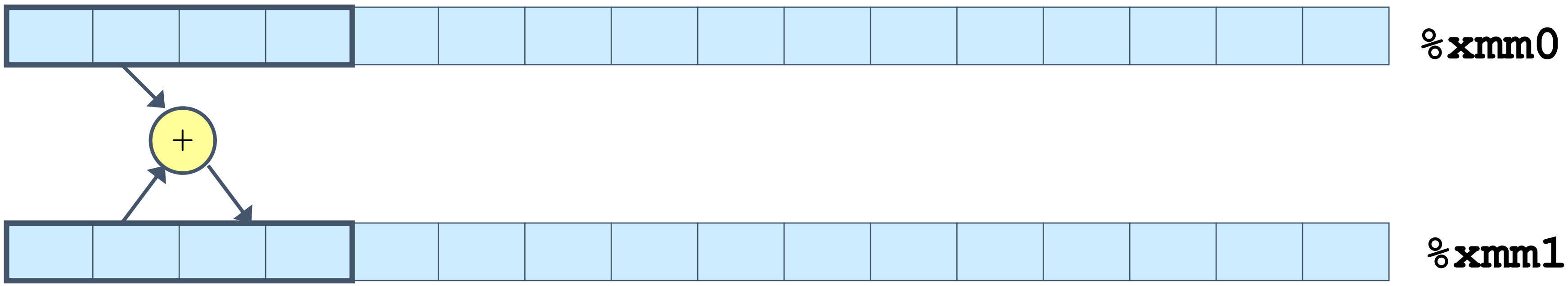
- 1 single-precision float



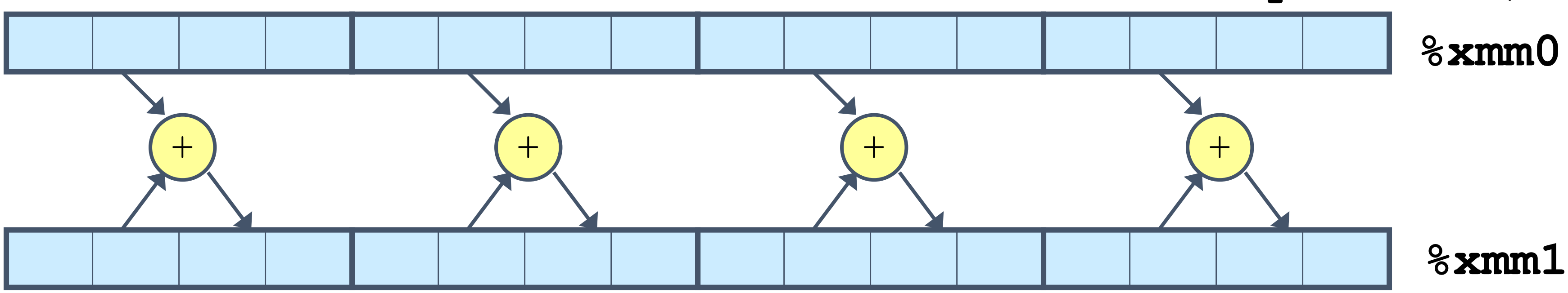
- 1 double-precision float

Scalar & SIMD Operations

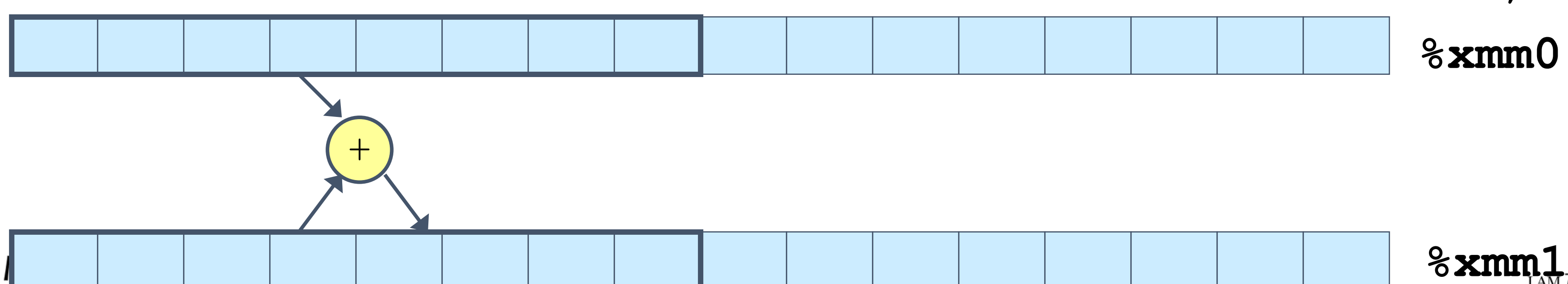
- Scalar Operations: Single Precision `addss %xmm0,%xmm1`



- SIMD Operations: Single Precision `addps %xmm0,%xmm1`



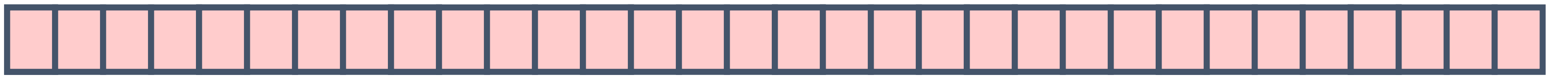
- Scalar Operations: Double Precision `addsd %xmm0,%xmm1`



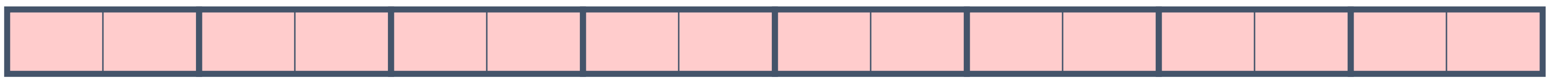
Programming with AVX2

YMM Registers

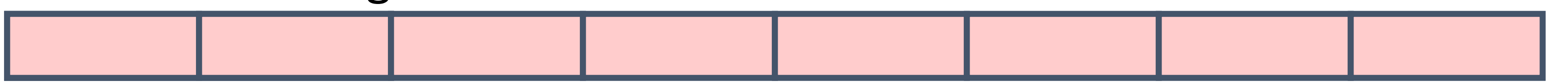
- 16 total, each 32 bytes
- 32 single-byte integers



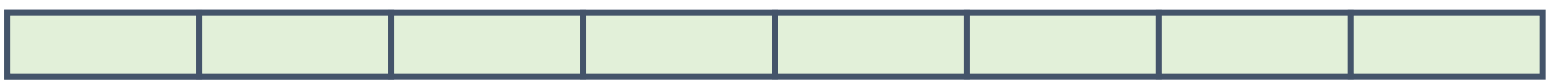
- 16 16-bit integers



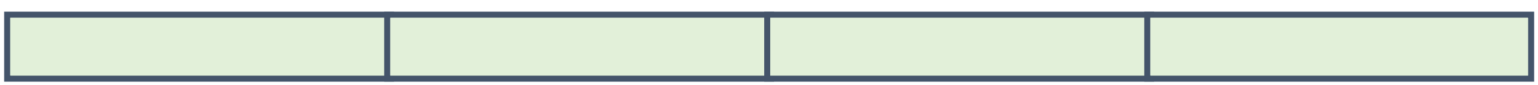
- 8 32-bit integers



- 8 single-precision floats



- 4 double-precision floats



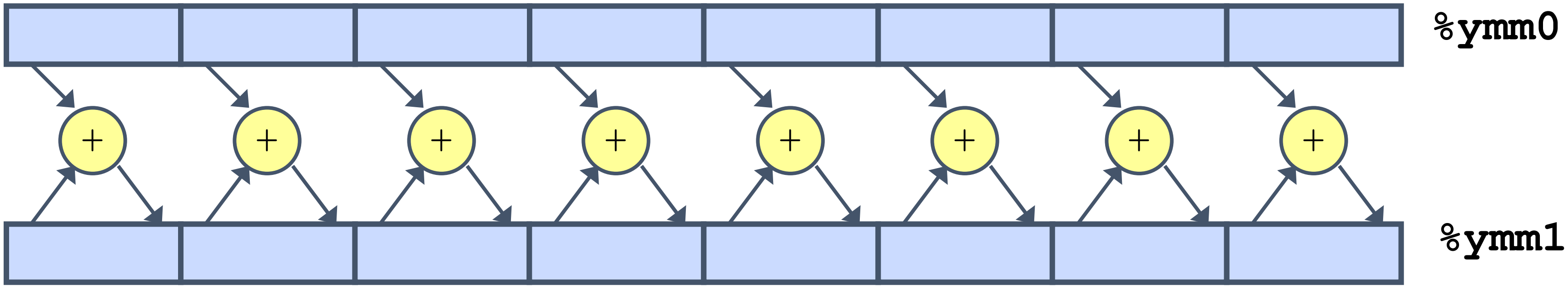
- 1 single-precision float

- 1 double-precision float

SIMD Operations (Single Instruction Multiple Data)

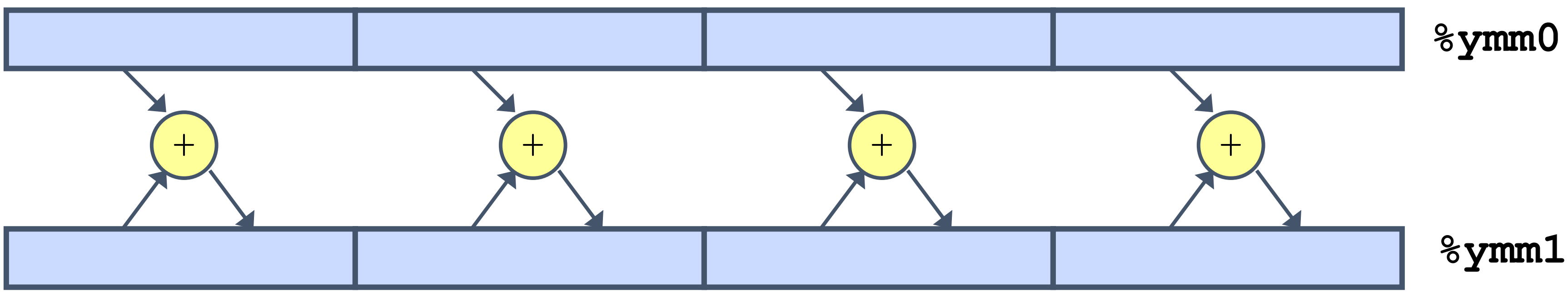
- SIMD Operations: Single Precision

```
vaddps %ymm0, %ymm1, %ymm1
```



- SIMD Operations: Double Precision

```
vaddpd %ymm0, %ymm1, %ymm1
```



Using Vector Instructions

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Scalar Best	0.54	1.01	1.01	0.52
Vector Best	0.06	0.24	0.25	0.16
Latency Bound	0.50	3.00	3.00	5.00
Throughput Bound	0.50	1.00	1.00	0.50
Vec Throughput Bound	0.06	0.12	0.25	0.12

- **Make use of AVX Instructions**
  - Parallel operations on multiple data elements
  - See OPT:SIMD

What About Branches?

- **Challenge**
  - Instruction Control Unit must work well ahead of Execution Unit to generate enough operations to keep EU busy

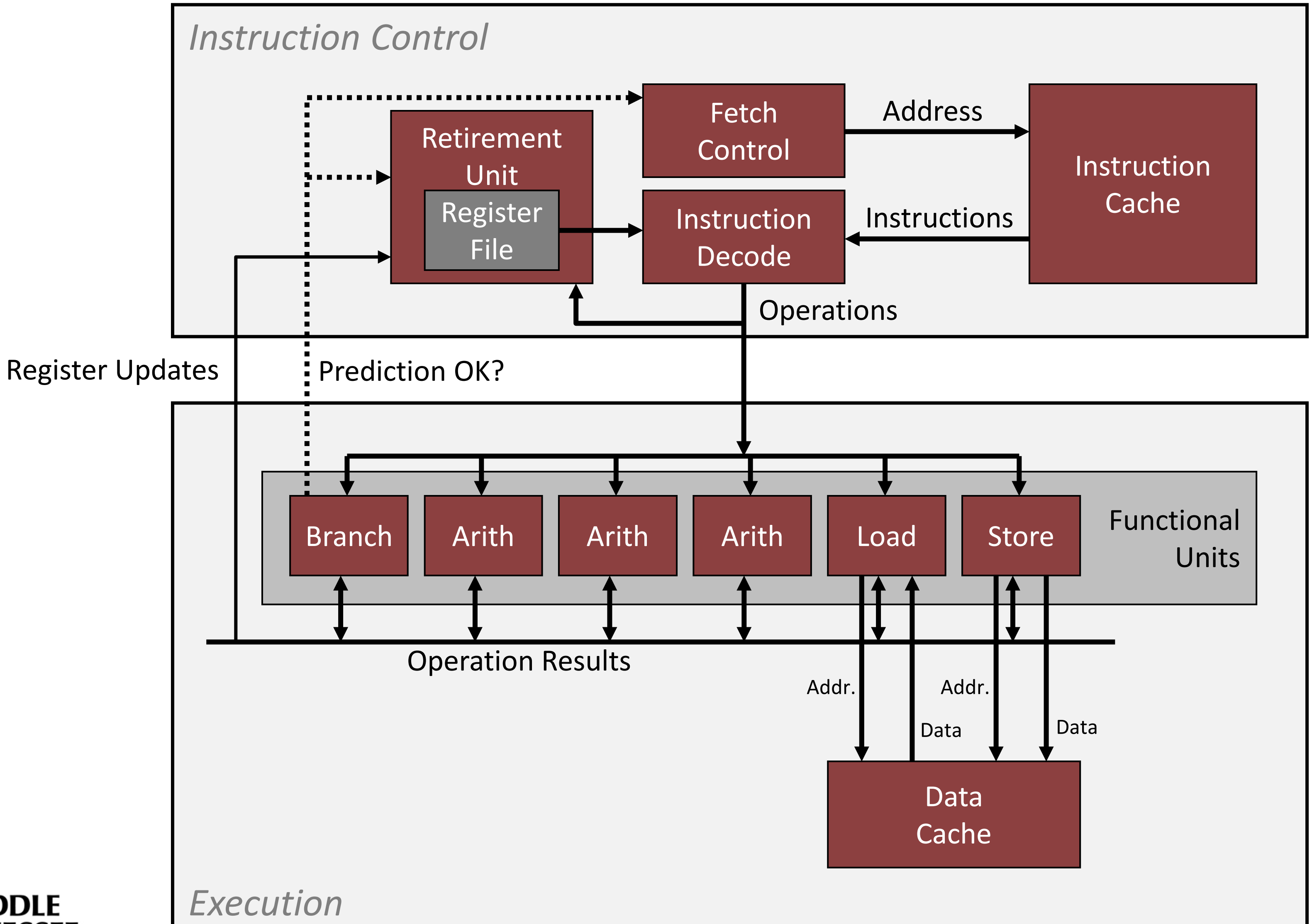
```
404663:  mov     $0x0,%eax
404668:  cmp     (%rdi),%rsi
40466b:  jge     404685
40466d:  mov     0x8(%rdi),%rax

 . . .

404685:  repz retq
```

Executing

How to continue?

  - When encounters conditional branch, cannot reliably determine where to continue fetching

Modern CPU Design



Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

Branch Outcomes

- **When encounter conditional branch, cannot determine where to continue fetching**
  - Branch Taken: Transfer control to branch target
  - Branch Not-Taken: Continue with next instruction in sequence
- **Cannot resolve until outcome determined by branch/integer unit**

```
404663:  mov     $0x0,%eax
404668:  cmp     (%rdi),%rsi
40466b:  jge     404685
40466d:  mov     0x8(%rdi),%rax

 . . .

404685:  repz retq
```

Branch Not-Taken

Branch Taken

Branch Prediction

- **Idea**
  - Guess which way branch will go
  - Begin executing instructions at predicted position
    - But don't actually modify register or memory data

```
  404663:  mov     $0x0,%eax
  404668:  cmp     (%rdi),%rsi
  40466b:  jge     404685
  40466d:  mov     0x8(%rdi),%rax

 . . .

  404685:  repz retq
```

Predict Taken

Begin Execution

Branch Prediction Through Loop

Assume
vector length = 100

```
401029:  vmulsd  (%rdx),%xmm0,%xmm0
40102d:  add     $0x8,%rdx
401031:  cmp     %rax,%rdx
401034:  jne     401029
```

i = 98

Predict Taken (OK)

```
401029:  vmulsd  (%rdx),%xmm0,%xmm0
40102d:  add     $0x8,%rdx
401031:  cmp     %rax,%rdx
401034:  jne     401029
```

i = 99

Predict Taken (Oops)

```
401029:  vmulsd  (%rdx),%xmm0,%xmm0
40102d:  add     $0x8,%rdx
401031:  cmp     %rax,%rdx
401034:  jne     401029
```

i = 100

Read invalid location

Executed

```
401029:  vmulsd  (%rdx),%xmm0,%xmm0
40102d:  add     $0x8,%rdx
401031:  cmp     %rax,%rdx
401034:  jne     401029
```

i = 101

Fetched

Branch Misprediction Invalidation

Assume
vector length = 100

```
401029:   vmulsd (%rdx),%xmm0,%xmm0
40102d:   add    $0x8,%rdx
401031:   cmp    %rax,%rdx
401034:   jne    401029
```
i = 98

Predict Taken (OK)

```
401029:   vmulsd (%rdx),%xmm0,%xmm0
40102d:   add    $0x8,%rdx
401031:   cmp    %rax,%rdx
401034:   jne    401029
```
i = 99

Predict Taken (Oops)

```
401029:   vmulsd (%rdx),%xmm0,%xmm0
40102d:   add    $0x8,%rdx
401031:   cmp    %rax,%rdx
401034:   jne    401029
```
i = 100

```
401029:   vmulsd (%rdx),%xmm0,%xmm0
40102d:   add    $0x8,%rdx
401031:   cmp    %rax,%rdx
401034:   jne    401029
```
i = 101

Invalidate

Branch Misprediction Recovery

```
401029:  vmulsd (%rdx),%xmm0,%xmm0
40102d:  add    $0x8,%rdx
401031:  cmp    %rax,%rdx
401034:  jne    401029
401036:  jmp    401040
  . . .
401040:  vmovsd %xmm0,(%r12)
```

i = 99

Definitely not taken

Reload Pipeline

- **Performance Cost**
  - Multiple clock cycles on modern processor
  - Can be a major performance limiter

Getting High Performance

- **Good compiler and flags**
- **Don't do anything stupid**
  - Watch out for hidden algorithmic inefficiencies
  - Write compiler-friendly code
    - Watch out for optimization blockers:
      procedure calls & memory references
  - Look carefully at innermost loops (where most work is done)

- **Tune code for machine**
  - Exploit instruction-level parallelism
  - Avoid unpredictable branches
  - Make code cache friendly (Covered later in course)