CS 33

Architecture and Optimization (2)

Most of the slides in this lecture are either from or adapted from slides provided by the authors of the textbook "Computer Systems: A Programmer's Perspective," 2nd Edition and are provided from the website of Carnegie-Mellon University, course 15-213, taught by Randy Bryant and David O'Hallaron in Fall 2010. These slides are indicated "Supplied by CMU" in the notes section of the slides.

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
- Most analysis is performed only within functions
  - 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

Supplied by CMU.

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];
```


Supplied by CMU.

In this example, we think of a as being a pointer to a matrix and we're copying array b into one row of a.

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 some Intel processors, multiplies are 3x longer than adds

- 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;
}
```


Supplied by CMU.

gcc does optimizations of the sort shown here.

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
```


Supplied by CMU.

gcc doesn't always figure out the best way to compile code. The code in the lower-left box is what gcc produced for the code in the upper left box. On the right is a much better version that was done by hand. (The C code was modified by hand; gcc then produced the better assembly code.)

Quiz 1

The fastest means for evaluating

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

requires exactly:

a) 2 multiplies and 2 additions
b) three additions
c) one multiply and two additions
d) one multiply and one addition

Optimization Blocker: Function Calls

- **Function to convert string to lower case**

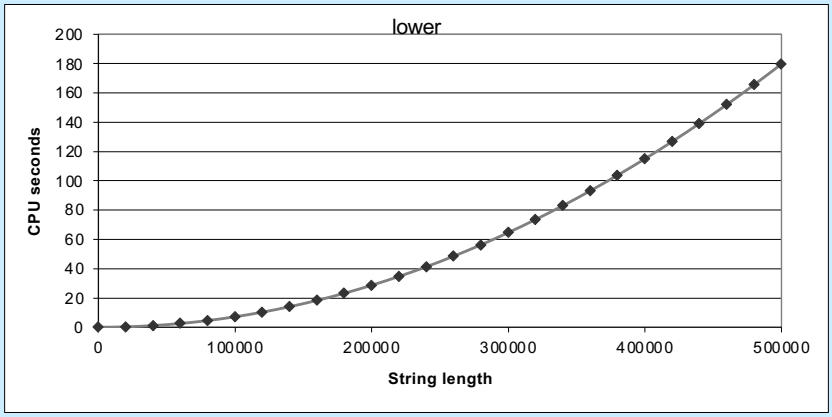
```
void lower(char *s){
  int i;
  for (i = 0; i < strlen(s); i++)
    if (s[i] >= 'A' && s[i] <= 'Z')
      s[i] -= ('A' - 'a');
}
```


Supplied by CMU.

Note that the expression (‘A’ - ‘a’) is a constant and is probably computed by the compiler itself.

Lower Case Conversion Performance

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



Supplied by CMU.

Convert Loop To Goto Form

```
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**

Supplied by CMU.

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
  - overall $O(N^2)$ performance

Supplied by CMU.

Improving Performance

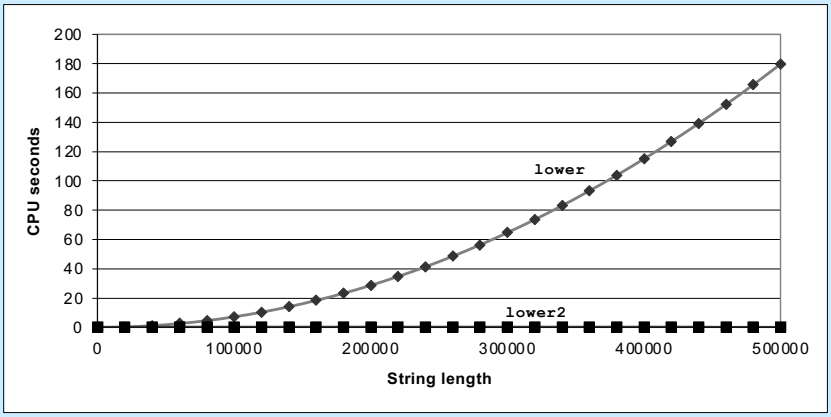
```
void lower2(char *s){
  int i;
  int len = strlen(s);
  for (i = 0; i < len; i++)
    if (s[i] >= 'A' && s[i] <= 'Z')
      s[i] -= ('A' - 'a');
}
```

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

Supplied by CMU.

Lower-Case Conversion Performance

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



Supplied by CMU.

The plot of lower2's performance looks flat (constant time), but it's actually linear – the slope is too small to appear non-zero in this plot.

Optimization Blocker: Function Calls

- *Why couldn't compiler move* `strlen` *out of inner loop?*
  - function 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
    - » function `lower` could interact with `strlen`
- **Warning:**
  - compiler treats function call as a black box
  - weak optimizations near them
- **Remedy:**
  - 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;
}
```


Supplied by CMU.

Memory Matters

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

```
# sum_rows1 inner loop
.L3:
   movq    (%r8,%rax,8), %rcx   # rcx = a[i][j]
   addq    %rcx, (%rdx)         # b[i] += rcx
   addq    $1, %rax             # j++
   cmpq    %rax, %rdi           # if i<n
   jne     .L3                  # goto .L3
```

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

Based on a slide supplied by CMU.

The issue here is whether it's really necessary to update the memory holding b[i] on every iteration of the inner for loop. Couldn't the value of b[i] be put in a register, updated there, then written to memory after the loop completes? Keep in mind that storing to memory is much more time-intensive than storing to a register.

Memory Aliasing

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

```
int A[3][3] =
  {{ 0,   1,   2},
   { 4,   8,  16},
   {32,  64, 128}};

int *B = &A[1][0];

sum_rows1(3, A, B;
```

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**

Supplied by CMU, updated for current gcc.

Removing Aliasing

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

```
# sum_rows2 inner loop
.L4:
   addq    (%r8, %rax, 8), %rcx
   addq    $1, %rax
   cmpq    %rax, %rdi
   jne     .L4
```

- **No need to store intermediate results**

Supplied by CMU.

Note that the programmer is implicitly assuming that the locations pointed to by a and b don't overlap.

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

Supplied by CMU.

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;
```


Pointers and Arrays

- `long a[][n]`
  - a is a 2-D array of longs, the size of each row is n
- `long (*c)[n]`
  - c is a pointer to a 1-D array of size n

- a and c are of the same type

Memory Matters, Fixed

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

```
# sum_rows1 inner loop
.L3:
   addq    (%rcx,%rax,8), %rdx
   addq    $1, %rax
   cmpq    %rax, %rdi
   jne     .L3
```

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

Note: we must give gcc the flag "-std=gnu99" for this to be compiled.

Observe that

`long (*a)[n]`

declares a to be a pointer to an array of n longs.

Thus

`long (*restrict a)[n]`

declares a to be a restricted pointer to an array of n longs

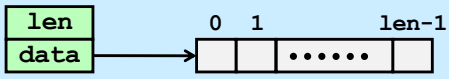
Exploiting Instruction-Level Parallelism

- 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

Supplied by CMU.

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;
}
```


Supplied by CMU.

Note that **get_vec_element** not only does an array lookup, but also does bounds checking.

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;
    }
}
```

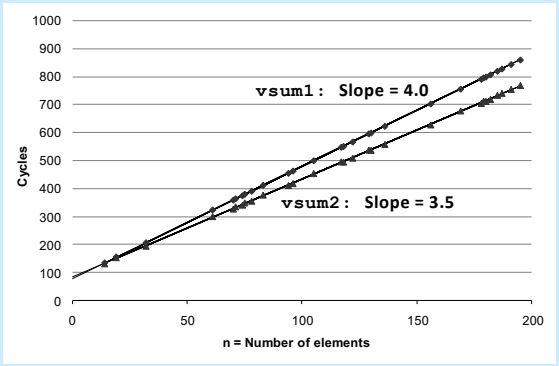
Compute sum or product of vector elements

- **Data Types**
  - **use different declarations for `data_t`**
    - » **`int`**
    - » **`float`**
    - » **`double`**
- **Operations**
  - **use different definitions of `OP` and `IDENT`**
    - » **`+, 0`**
    - » **`*, 1`**

Supplied by CMU.

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



Supplied by CMU.

A **cycle** is a measure of processor time, often referred to as a **clock cycle**. Processors are driven by a clock, running at a certain frequency, say 10 GHz ($10*2^{30}$ cycles per second). In this case, the length of a cycle is the period of the clock (the reciprocal of its frequency -- $.1*2^{-30}$ seconds).

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

Supplied by CMU.

The times given in the table are in cycles/element. The unoptimized code was compiled with the –O0 flag. The code would most likely be faster if compiled with the –O2 flag, but the purpose of these slides is to figure out exactly what can make it run faster.

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

Supplied by CMU.

Since the result of calling **vec_length** never changes, for the given vector v, there's no point to calling it in every iteration of the loop. So, we move it out of the loop and call it just once, with dramatic improvement of performance.

Eliminate Function Calls

```
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];
    }
}
```

```
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

Supplied by CMU.

Since bounds checking isn't necessary, we replace **get_vec_element** with a simple array lookup.

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

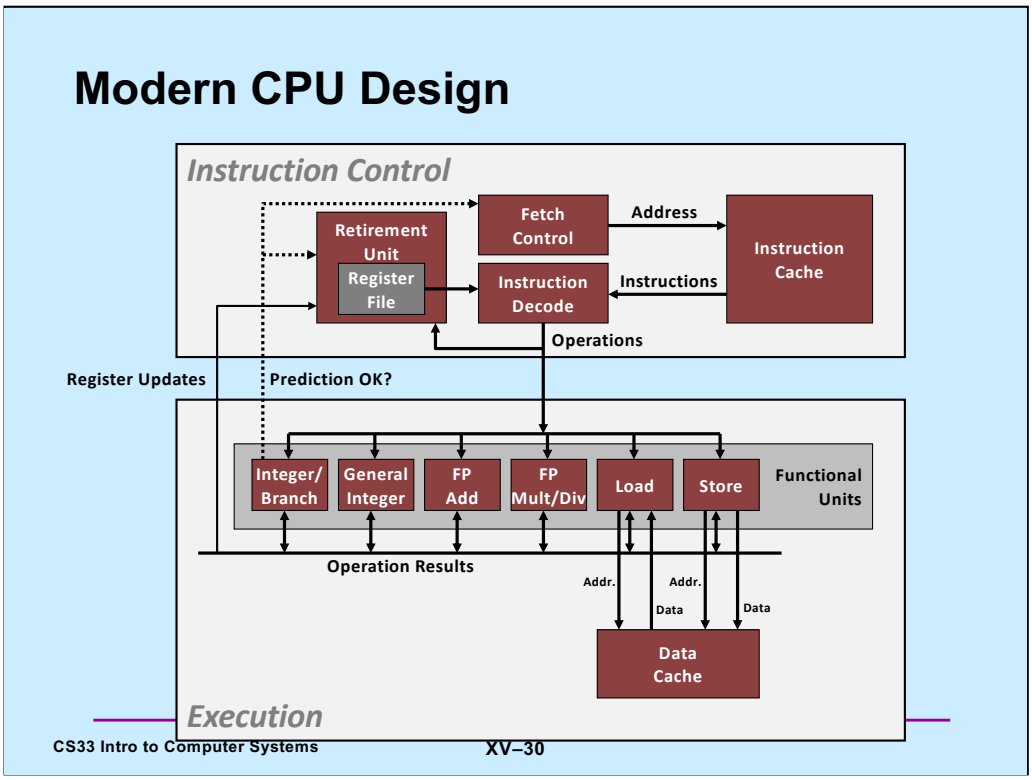
Supplied by CMU.

Finally, we recognize that we don't need to update ***dest** on each iteration, but only when we're done.

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×
d) 128× – ∞×



Supplied by CMU.

Multiple Operations per Instruction

- addq %rax, %rdx
  - a single operation
- addq %rax, 8(%rdx)
  - three operations
    - » load value from memory
    - » add to it the contents of %rax
    - » store result in memory

Instruction-Level Parallelism

- `addq 8(%rax), %rax`
  `addq %rbx, %rdx`
  - can be executed simultaneously: completely independent
- `addq 8(%rax), %rbx`
  `addq %rbx, %rdx`
  - can also be executed simultaneously, but some coordination is required

Out-of-Order Execution

```
• movss    (%rbp), %xmm0
  mulss    (%rax, %rdx, 4), %xmm0
  movss    %xmm0, (%rbp)
  addq     %r8, %r9
  imulq    %rcx, %r12
  addq     $1, %rdx
```

these can be executed without waiting for the first three to finish

Note that the first three instructions are floating-point instructions, and %xmm0 is a floating-point register.

Speculative Execution

```
80489f3:    movl    $0x1,%ecx
80489f8:    xorq    %rdx,%rdx
80489fa:    cmpq    %rsi,%rdx
80489fc:    jnl     8048a25
80489fe:    movl    %esi,%edi
8048a00:    imull   (%rax,%rdx,4),%ecx
```

perhaps execute these instructions

Haswell CPU

- **Functional Units**
  1) **Integer arithmetic, floating-point multiplication, integer and floating-point division, branches**
  2) **Integer arithmetic, floating-point addition, integer and floating-point multiplication**
  3) **Load, address computation**
  4) **Load, address computation**
  5) **Store**
  6) **Integer arithmetic**
  7) **Integer arithmetic, branches**
  8) **Store, address computation**

Supplied by CMU.

"Haswell" is Intel's code name for relatively recent versions of its Core I7 and Core I5 processor design. Most of the computers in Brown CS employ Core I5 processors.

While Apple's M1 and M2 processors have a different architecture, producing code for them involves similar concerns as producing code for Haswell processors.

Haswell CPU

- **Instruction characteristics**

Instruction	*Latency*	*Cycles/Issue*	*Capacity*
Integer Add	1	1	4
Integer Multiply	3	1	1
Integer/Long Divide	3-30	3-30	1
Single/Double FP Add	3	1	1
Single/Double FP Multiply	5	1	2
Single/Double FP Divide	3-15	3-15	1
Load	4	1	2
Store	-	1	2

Supplied by CMU.

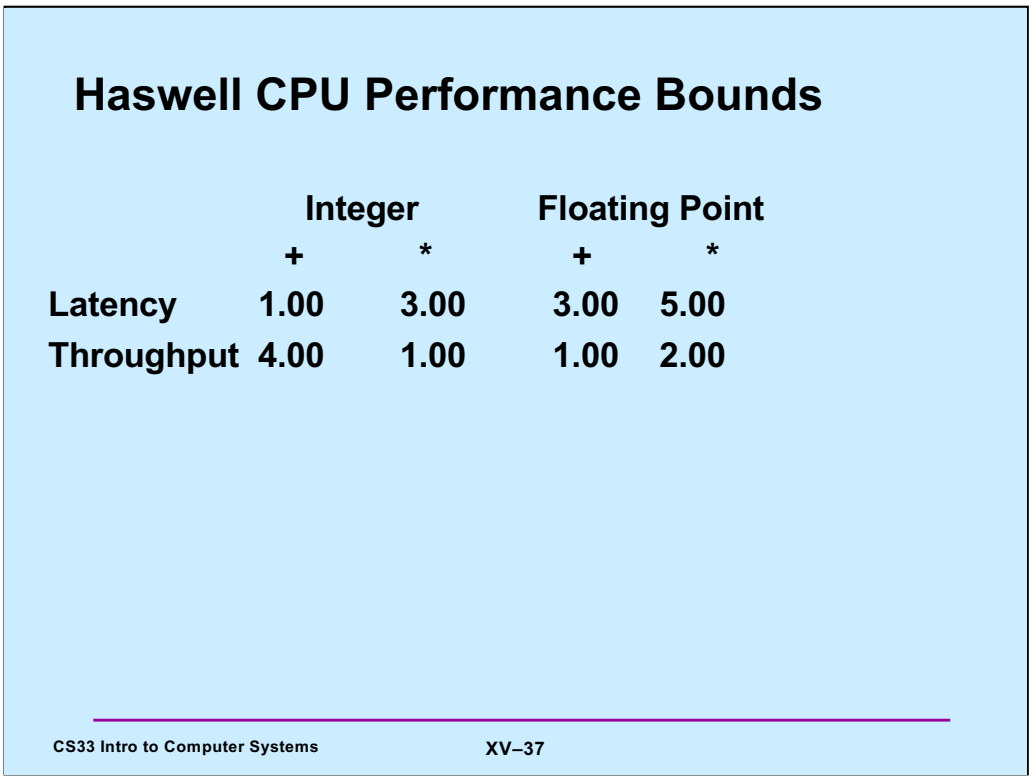
These figures are for those cases in which the operands are either in registers or are immediate. For the other cases, additional time is required to load operands from memory or store them to memory.

"Cycles/Issue" is the number of clock cycles that must occur from the start of execution of one instruction to the start of execution to the next. The reciprocal of this value is the throughput: the number of instructions (typically a fraction) that can be completed per cycle.

"Capacity" is the number of functional units that can do the indicated operations.

The figures for load and store assume the data is coming from/going to the data cache. Much more time is required if the source or destination is RAM.

The latency for stores is a bit complicated – we might discuss it in a later lecture.



Derived from a slide provided by CMU.

We assume that the source and destination are either immediate (source only) or registers. Thus, any bottlenecks due to memory access do not arise.

Each integer add requires one clock cycle of latency. It's also the case that, for each functional unit doing integer addition, the time required between add instructions is one clock cycle. However, since there are four such functional units, all four can be kept busy with integer add instructions and thus the aggregate throughput can be as good as one integer add instruction completing, on average, every .25 clock cycles, for a throughput of 4 instructions/cycle.

Each integer multiply requires three clock cycles. But since a new multiply instruction can be started every clock cycle (i.e., they can be pipelined), the aggregate throughput can be as good as one integer multiply completing every clock cycle.

Each floating point multiply requires five clock cycles, but they can be pipelined with one starting every clock cycle. Since there are two functional units that can perform floating point multiply, the aggregate throughput can be as good as one completing every .5 clock cycles, for a throughput of 2 instructions/cycle.

x86-64 Compilation of Combine4

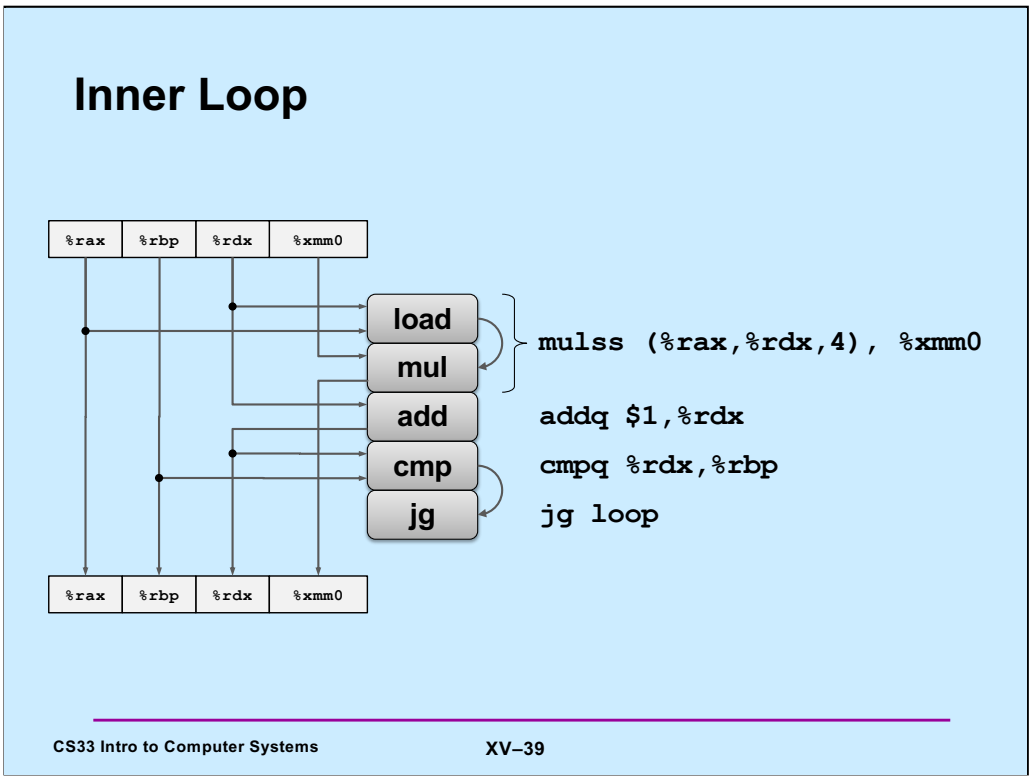
- **Inner loop (case: SP floating-point multiply)**

```
.L519:                          # Loop:
  mullss (%rax,%rdx,4), %xmm0 # t = t * d[i]
  addq  $1, %rdx                # i++
  cmpq  %rdx, %rbp              # Compare length:i
  jg    .L519                   # If >, goto Loop
```

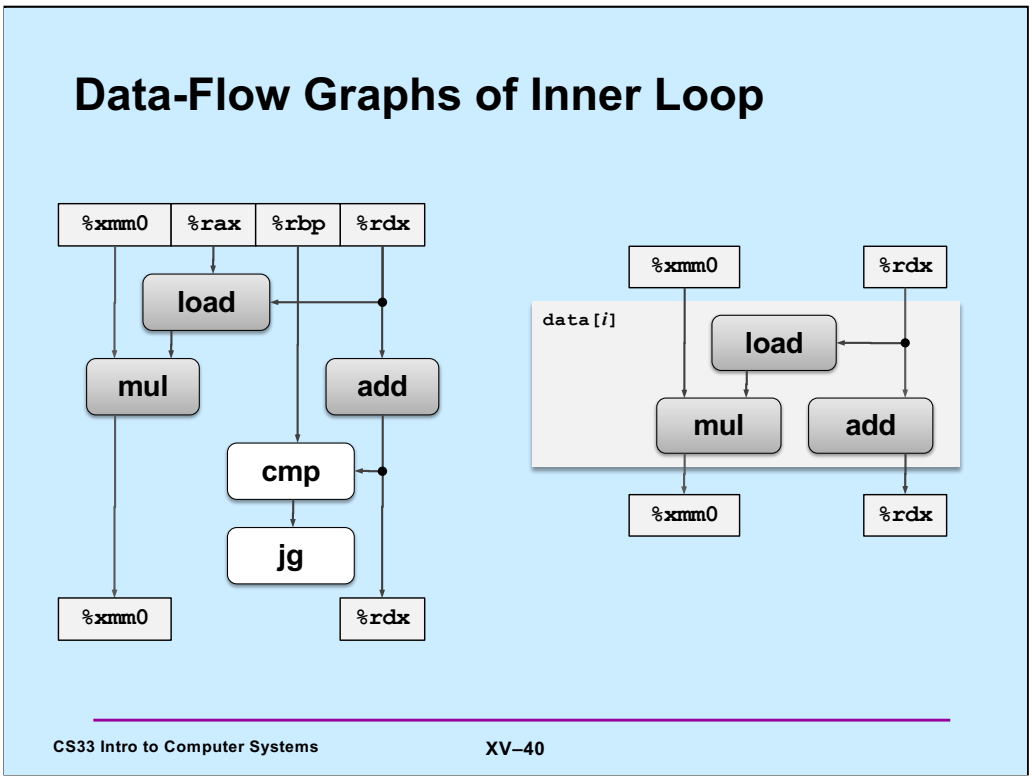
Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	1.27	3.00	3.00	5.00
Latency bound	1.00	3.00	3.00	5.0
Throughput bound	0.25	1.00	1.00	0.50

Supplied by CMU.

These numbers are for the Haswell CPU. The row labelled "Combine4" gives the actual time, in clock cycles, taken by each execution of the loop. The row labelled "Latency bound" gives the time required for the arithmetic instruction (integer add or multiply, double-precision floating-point add or multiply) in each execution of the loop. The last row, "Throughput bound", gives the time required for the arithmetic instructions if they can be executed without delays by the multiple execution units – i.e., there are no data hazards (as explained in the previous lecture).



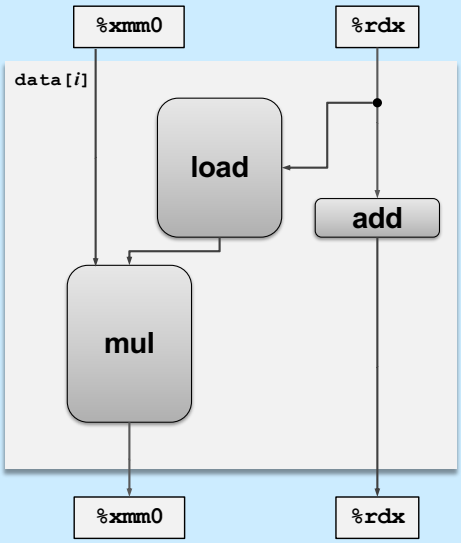
This is Figure 5.13 of Bryant and O'Hallaron. It shows the code for the single-precision floating-point version of our example.



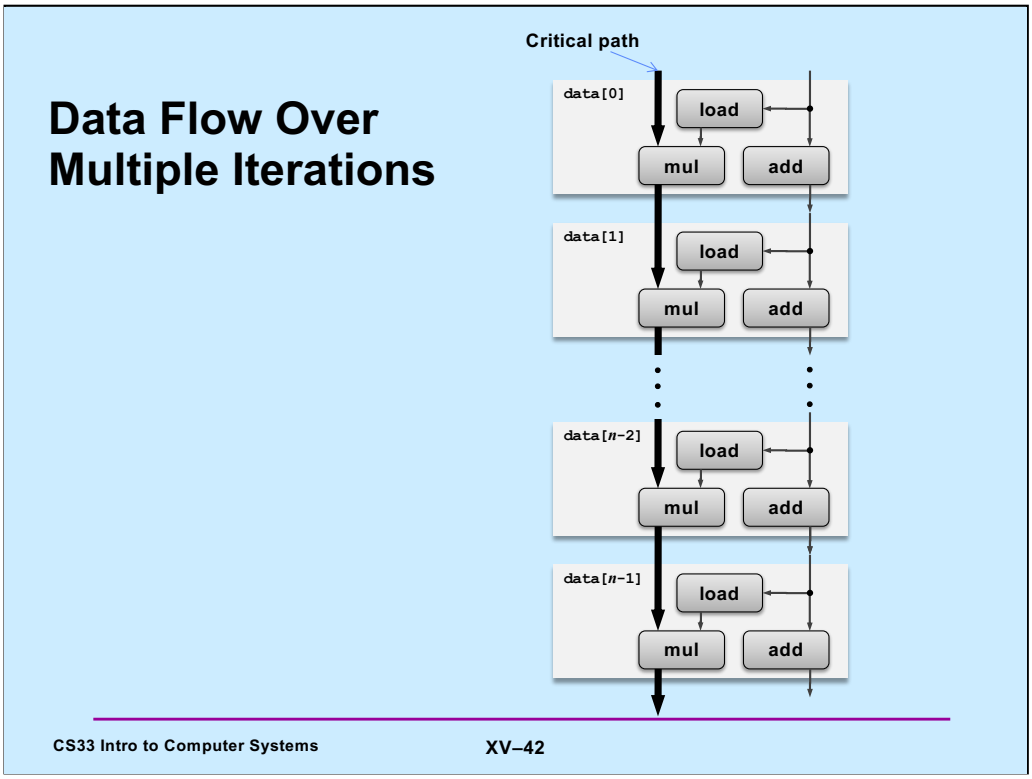
These are Figures 5.14 a and b of Bryant and O'Hallaron.

Since the values in %rax and %rbp don't change during the execution of the inner loop, they're not critical to the scheduling and timing of the instructions. Assuming the branch is taken, the **cmp** and **jg** instructions also aren't a factor in determining the timing of the instructions. We focus on what's shown in the righthand portion of the slide.

Relative Execution Times

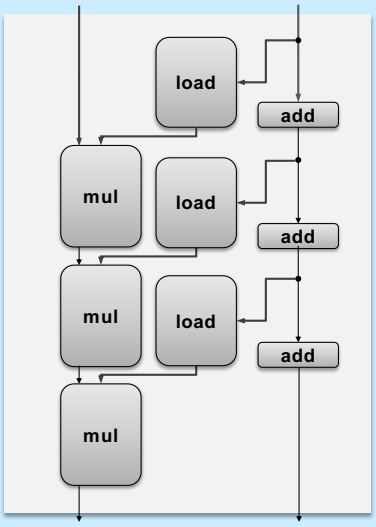


Here we modify the graph of the previous slide to show the relative times required of **mul**, **load**, and **add**.



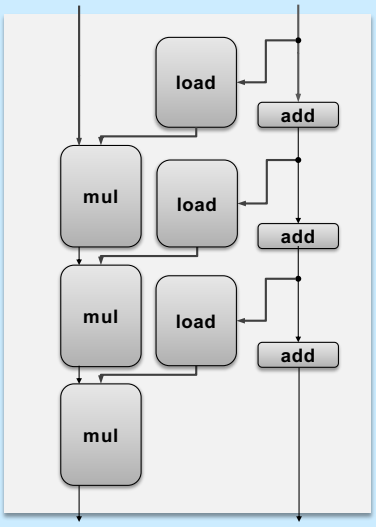
This is Figure 5.15 of Bryant and O'Hallaron.

Pipelined Data-Flow Over Multiple Iterations



Without pipelining, the data flow would appear as shown in the slide.

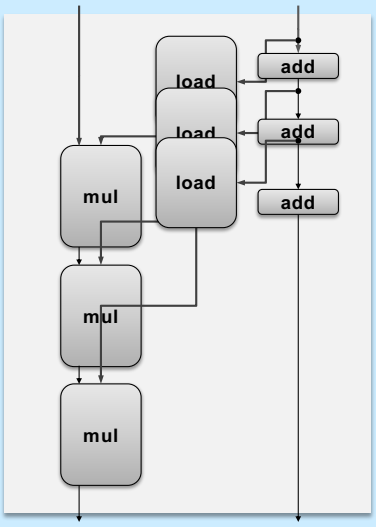
Pipelined Data-Flow Over Multiple Iterations



The loads depend only on the computation of the array index, which is quickly done by addition units. Thus, the loads can be pipelined.

It's clear that the multiplies form the critical path, since they use the results of the previous multiplies.

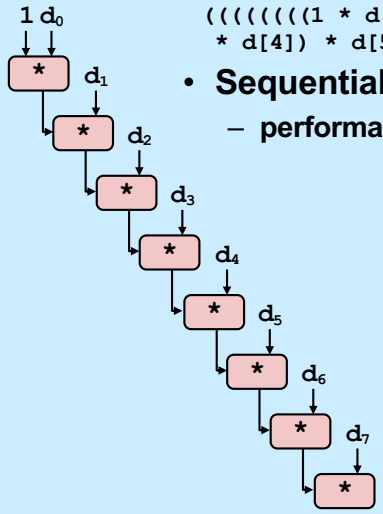
Pipelined Data-Flow Over Multiple Iterations



The loads depend only on the computation of the array index, which is quickly done by addition units. Thus, the loads can be pipelined.

It's clear that the multiplies form the critical path, since they use the results of the previous multiplies.

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**

Supplied by CMU.

Since the multiplies form the critical path, here we focus only on them. In what's shown here, only one multiply can be done at a time, since the result of the one multiply is needed for the next.

Loop Unrolling

```
void unroll2x(vec_ptr_t v, data_t *dest)
{
    int length = vec_length(v);
    int limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x = IDENT;
    int 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**

Supplied by CMU.

Effect of Loop Unrolling

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	1.27	3.00	3.00	5.00
Unroll 2x	1.01	3.00	3.00	5.00
Latency bound	1.0	3.0	3.0	5.0
Throughput bound	0.25	1.0	1.0	0.5

- Helps integer add
  - reduces loop overhead
- Others don't improve. *Why?*
  - still sequential dependency

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


Supplied by CMU.

Loop Unrolling with Reassociation

```
void unroll2xra(vec_ptr_t v, data_t *dest)
{
    int length = vec_length(v);
    int limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x = IDENT;
    int 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

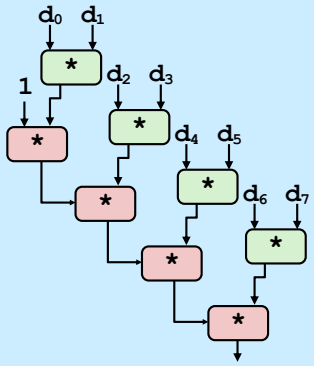
```
x = (x OP d[i]) OP d[i+1];
```

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

Supplied by CMU.

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
  - should be (N/2+1)*D cycles: CPE = D/2
  - measured CPE slightly worse for integer addition (there are other things going on)

Supplied by CMU.

How much time is required to compute the products shown in the slide? The multiplications in the upper right of the tree, directly involving the d_i, could all be done at once, since there are no dependencies; thus, computing them can be done in D cycles, where D is the latency required for multiply. This assumes we have a sufficient number of functional units to do this, thus this is a lower bound. The multiplications in the lower left must be done sequentially, since each depends on the previous; thus, computing them requires (N/2)*D cycles. Since first of the top right multiplies must be completed before the bottom left multiplies can start, the overall performance has a lower bound of (N/2 + 1)*D.

Effect of Reassociation

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	1.27	3.00	3.00	5.00
Unroll 2x	1.01	3.00	3.00	5.00
Unroll 2x, reassociate	1.01	1.51	1.51	2.51
Latency bound	1.0	3.0	3.0	5.0
Throughput bound	.25	1.0	1.0	.5

- Nearly 2x speedup for int *, FP +, FP *
  - reason: breaks sequential dependency

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


Supplied by CMU.

Loop Unrolling with Separate Accumulators

```
void unroll2xp2x(vec_ptr_t v, data_t *dest)
{
    int length = vec_length(v);
    int limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x0 = IDENT;
    data_t x1 = IDENT;
    int 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**

Supplied by CMU.

Here one "accumulator" (x0) is summing the array elements with even indices, the other (x1) is summing array elements with odd indices.

Effect of Separate Accumulators

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	1.27	3.00	3.00	5.00
Unroll 2x	1.01	3.00	3.00	5.00
Unroll 2x, reassociate	1.01	1.51	1.51	2.01
Unroll 2x parallel 2x	.81	1.51	1.51	2.51
Latency bound	1.0	3.0	3.0	5.0
Throughput bound	.25	1.0	1.0	.5

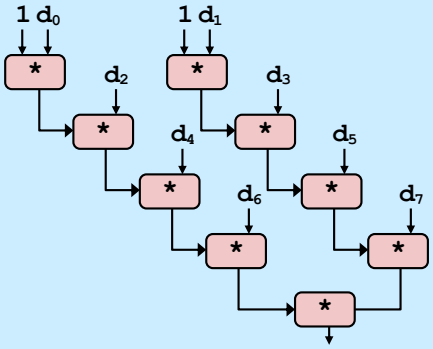
- 2x speedup (over unroll 2x) for int *, FP +, FP *
  - breaks sequential dependency in a "cleaner," more obvious way

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


Supplied by CMU.

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**
  - Integer addition improved, but not yet at predicted value

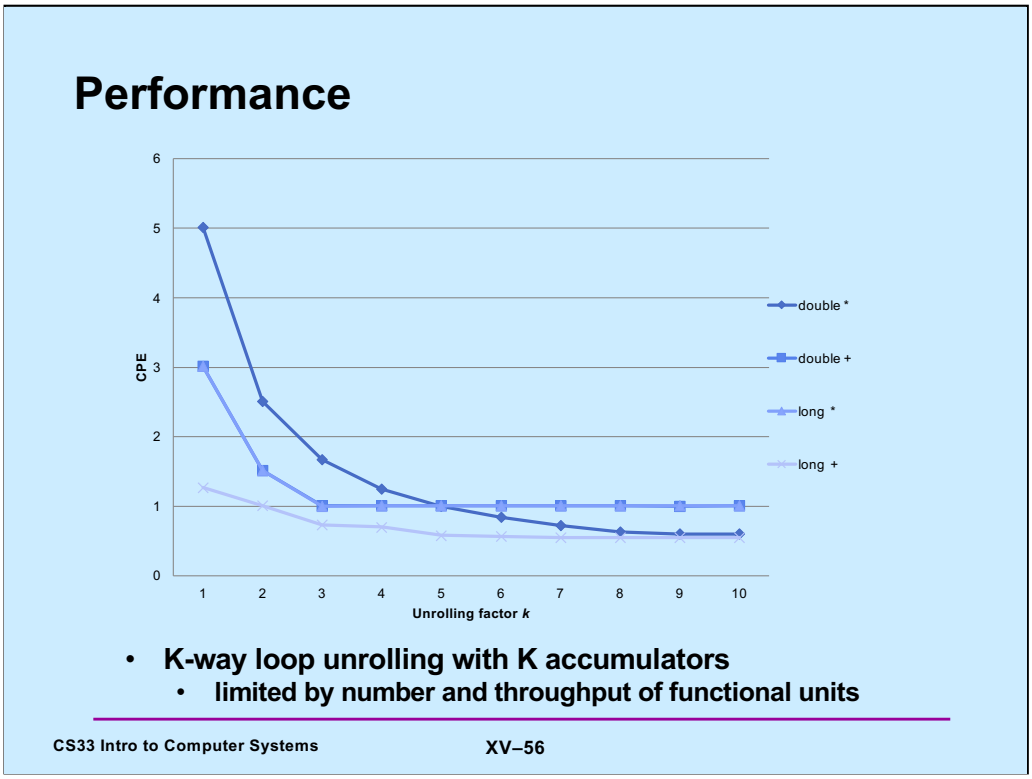
What Now?

Supplied by CMU.

Quiz 3

We're making progress. With two accumulators we get a two-fold speedup. With three accumulators, we can get a three-fold speedup. How much better performance can we expect if we add even more accumulators?

a) It keeps on getting better as we add more and more accumulators
b) It's limited by the latency bound
c) It's limited by the throughput bound
d) It's limited by something else



This is Figure 5.30 from the textbook.

Achievable Performance

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	1.27	3.0	3.0	5.0
Achievable scalar	.52	1.01	1.01	.54
Latency bound	1.00	3.00	3.00	5.00
Throughput bound	.25	1.00	1.00	.5

Based on a slide supplied by CMU.

Using Vector Instructions

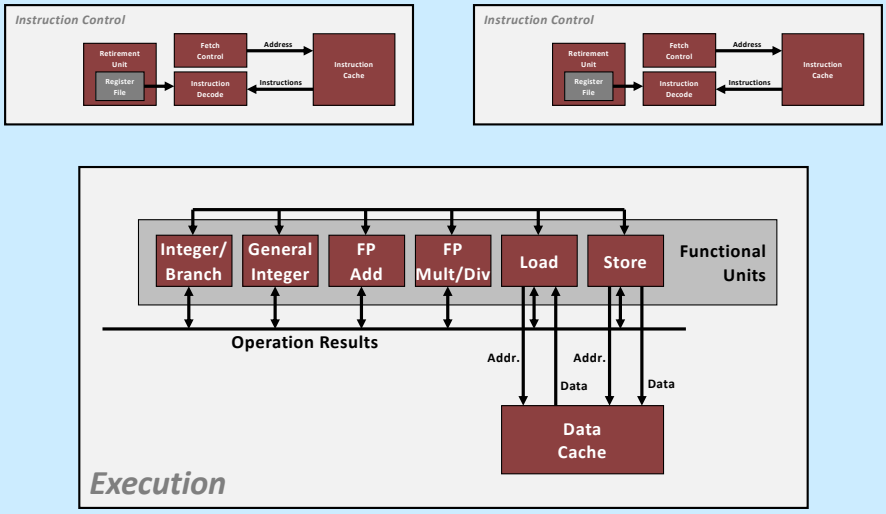
Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	1.27	3.0	3.0	5.0
Achievable Scalar	.52	1.01	1.01	.54
Latency bound	1.00	3.00	3.00	5.00
Throughput bound	.25	1.00	1.00	.5
Achievable Vector	.05	.24	.25	.16
Vector throughput bound	.06	.12	.25	.12

- **Make use of SSE Instructions**
  - **parallel operations on multiple data elements**

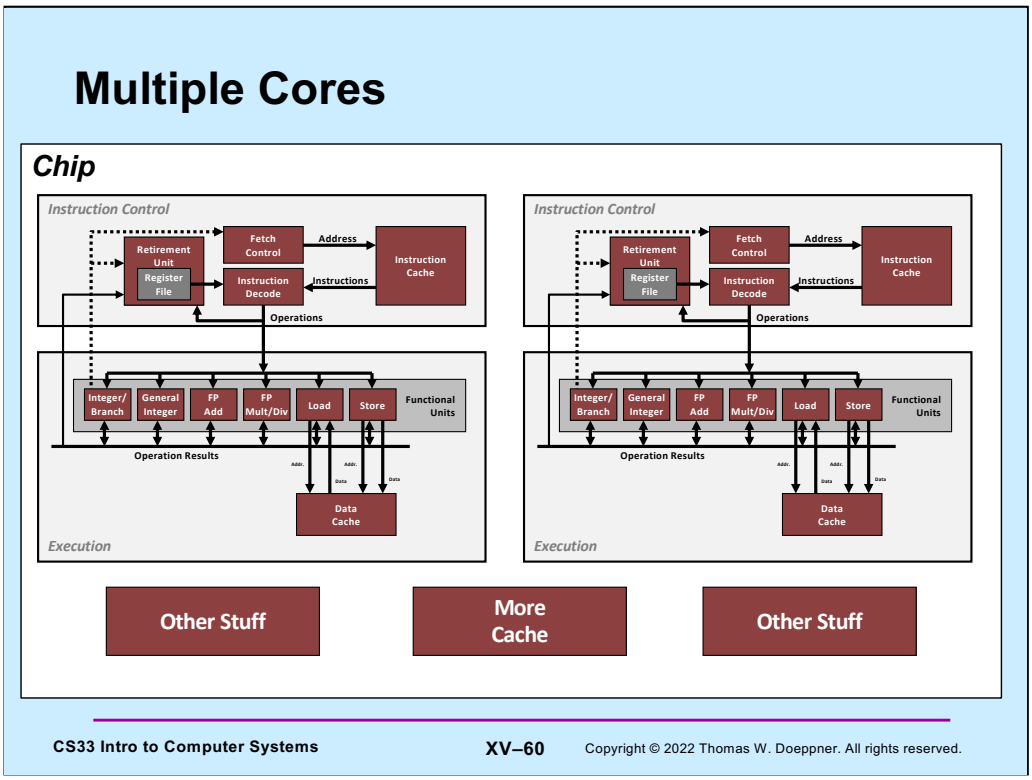
Based on a slide supplied by CMU.

SSE stands for “streaming SIMD extensions”. SIMD stands for “single instruction multiple data” – these are instructions that operate on vectors.

Hyper Threading



One way of improving the utilization of the functional units of a processor is hyperthreading. The processor supports multiple instruction streams ("hyper threads"), each with its own instruction control. But all the instruction streams share the one set of functional units.



Going a step further, one can pack multiple complete processors onto one chip. Each processor is known as a core and can execute instructions independently of the other cores (each has its private set of functional units). In addition to each core having its own instruction and data cache, there are caches shared with the other cores on the chip. We discuss this in more detail in a subsequent lecture.

In many of today's processor chips, hyperthreading is combined with multiple cores. Thus, for example, a chip might have four cores each with four hyperthreads. Thus, the chip might handle 16 instruction streams.