How to Write Fast Numerical Code

Spring 2011

Lecture 4

Instructor: Markus Püschel

TA: Georg Ofenbeck

ETH

Eidgenössische Technische Hochschule Zürich

Swiss Federal Institute of Technology Zurich

Organizational

- **Class Monday 14.3. → Friday 18.3**
- **Office hours:**
  - Markus: Tues 14–15:00
  - Georg: Wed 14–15:00
- **Research projects**

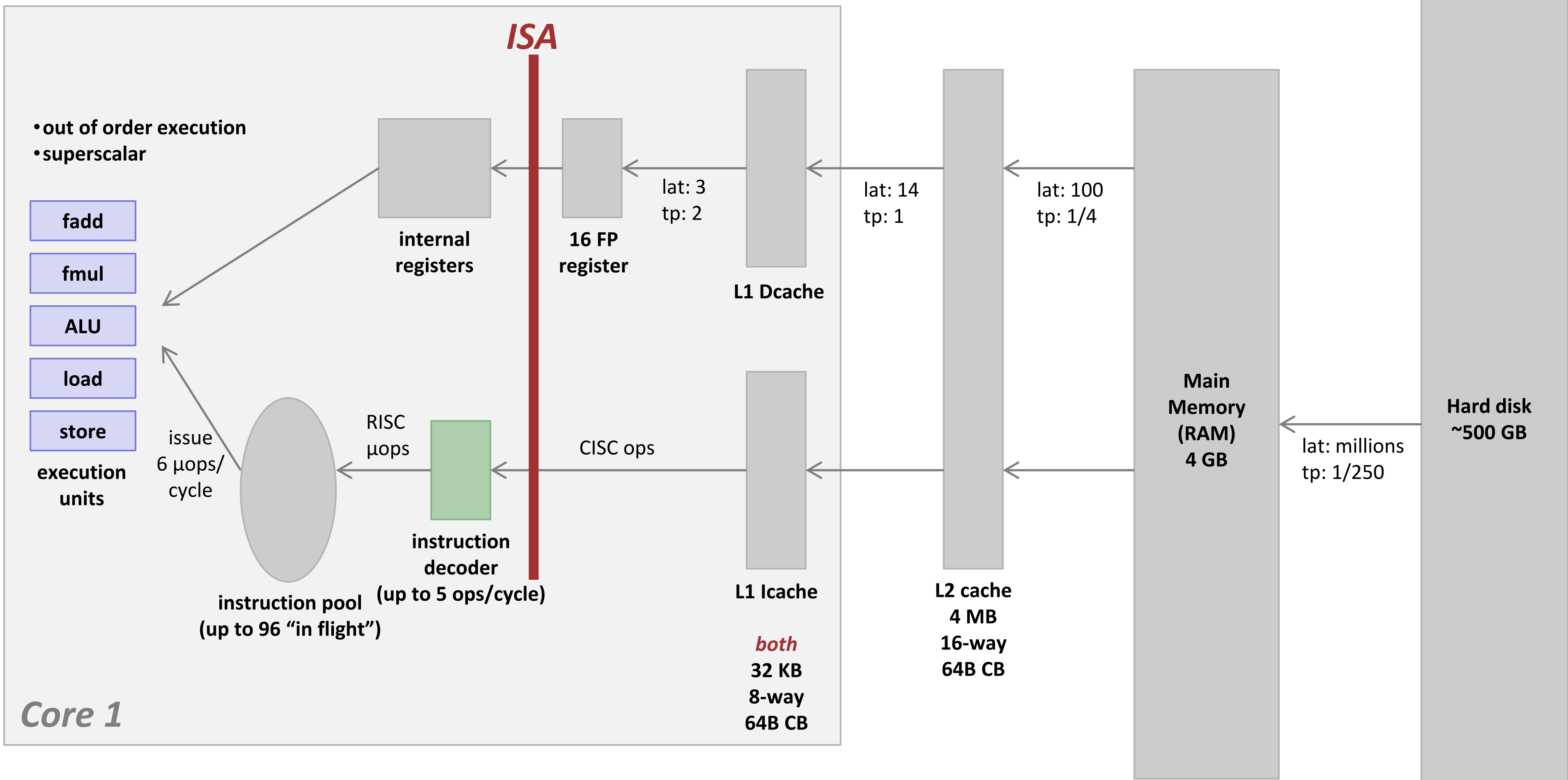
Abstracted Microarchitecture: Example Core (2008)

Throughput is measured in doubles/cycle

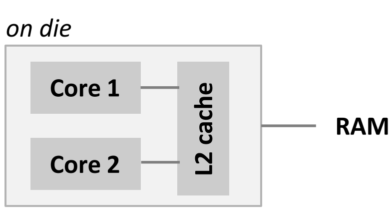
Latency in cycles for one double

1 double = 8 bytes

Rectangles not to scale



Core 2 Duo:



Memory hierarchy:

- Registers
- L1 cache
- L2 cache
- Main memory
- Hard disk

Organization

- **Instruction level parallelism (ILP): an example**
- **Optimizing compilers and optimization blockers**

Chapter 5 in ***Computer Systems: A Programmer's Perspective****, 2nd edition, Randal E. Bryant and David R. O'Hallaron, Addison Wesley 2010*

Core 2: Instruction Decoding and Execution Units

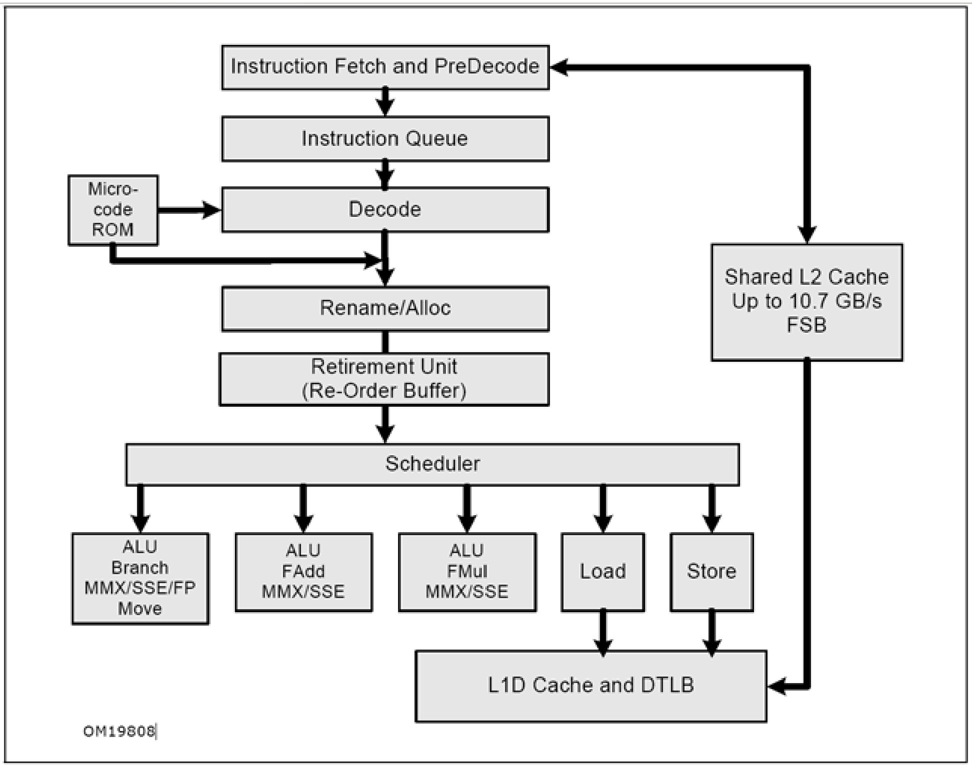


Figure 2-1. Intel Core Microarchitecture Pipeline Functionality

Latency/throughput (double)

FP Add: 3, 1

FP Mult: 5, 1

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 CPUs since about 1998 are superscalar
- Intel: since Pentium Pro

Hard Bounds: Pentium 4 vs. Core 2

- **Pentium 4 (Nocona)**

Instruction	*Latency*	*Cycles/Issue*
Load / Store	5	1
Integer Multiply	10	1
Integer/Long Divide	36/106	36/106
Single/Double FP Multiply	**7**	**2**
Single/Double FP Add	**5**	**2**
Single/Double FP Divide	32/46	32/46

- **Core 2**

Instruction	*Latency*	*Cycles/Issue*
Load / Store	5	1
Integer Multiply	3	1
Integer/Long Divide	18/50	18/50
Single/Double FP Multiply	**4/5**	**1**
Single/Double FP Add	**3**	**1**
Single/Double FP Divide	18/32	18/32

Hard Bounds (cont'd)

- **How many cycles at least if**
  - Function requires n float adds?
  - Function requires n float ops (adds and mults)?
  - Function requires n int mults?

Performance in Numerical Computing

- **Numerical computing =**
  computing dominated by floating point operations
- **Example: Matrix multiplication**
- **Performance measure (in most cases) for a numerical function:**

$$\frac{\#\textit{floating point operations}}{\textit{runtime [s]}}$$

- **Theoretical peak performance on 3 GHz Core 2 (1 core)?**
  - Scalar (no SSE): ***6 Gflop/s***
  - SSE double precision: ***12 Gflop/s***
  - SSE single precision: ***24 Gflop/s***

Example Computation (on Pentium 4)

```
void combine4(vec_ptr 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;
}
```

`d[0] OP d[1] OP d[2] OP … OP d[length-1]`

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

Runtime of Combine4 (Pentium 4)

- **Use cycles/OP**

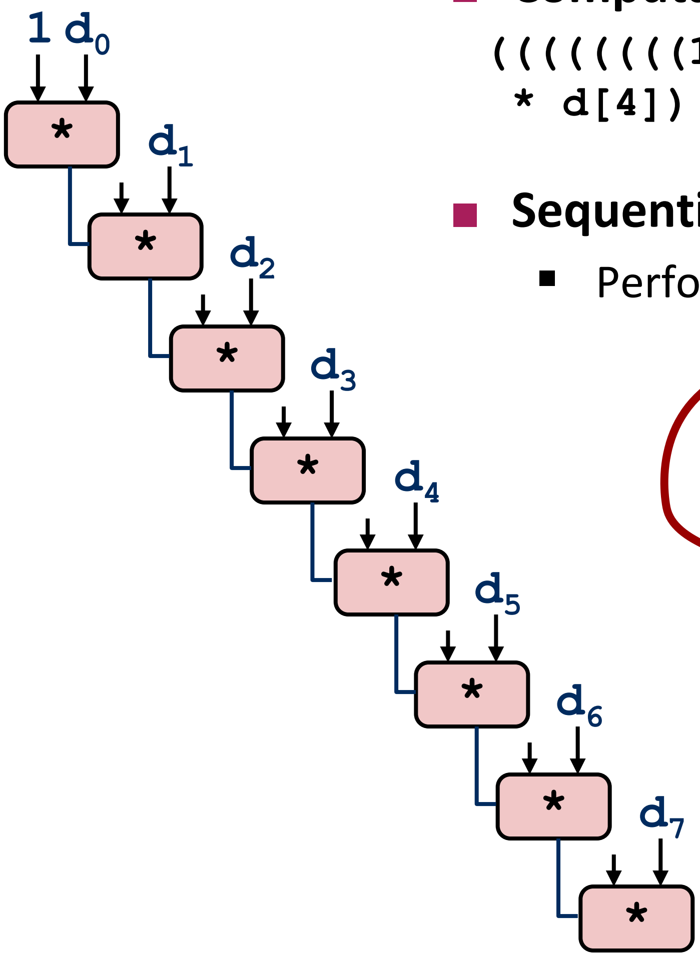
```
void combine4(vec_ptr v,
  data_t *dest)
{
  int i;
  int length = vec_length(v);
  data_t *d = get_vec_start(v);
  data_t t = IDENT;
  for (i = 0; i < length; i++)
    t = t OP d[i];
  *dest = t;
}
```

Cycles per OP

Method	Int (add/mult)		Float (add/mult)	
combine4	2.2	10.0	5.0	7.0
bound	1.0	1.0	2.0	2.0

- **Questions:**
  - Explain red row
  - Explain gray row

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 = no ILP! Hence,**
  - Performance: determined by latency of OP!

Cycles per element (or per OP)

Method	Int (add/mult)		Float (add/mult)	
combine4	2.2	10.0	5.0	7.0
bound	1.0	1.0	2.0	2.0

Loop Unrolling

```
void unroll2(vec_ptr 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

Effect of Loop Unrolling

Method	Int (add/mult)		Float (add/mult)	
combine4	2.2	10.0	5.0	7.0
unroll2	1.5	10.0	5.0	7.0
bound	1.0	1.0	2.0	2.0

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

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

Loop Unrolling with Reassociation

```
void unroll2_ra(vec_ptr 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;
}
```

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

Effect of Reassociation

Method	Int (add/mult)		Float (add/mult)	
combine4	2.2	10.0	5.0	7.0
unroll2	1.5	10.0	5.0	7.0
unroll2-ra	1.56	5.0	2.75	3.62
bound	1.0	1.0	2.0	2.0

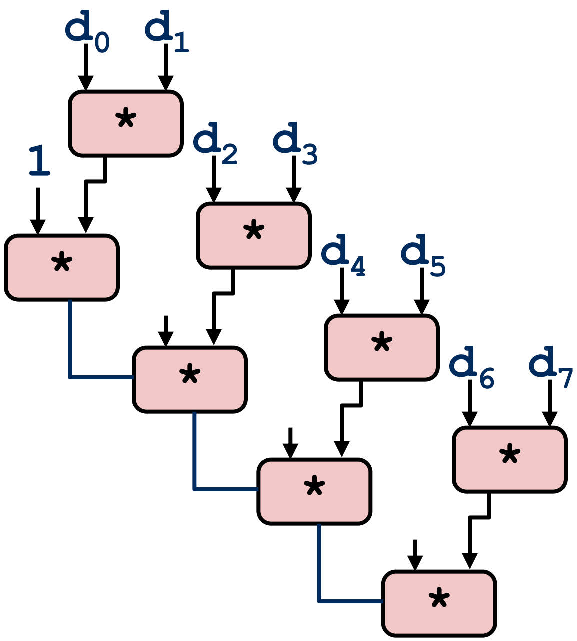
- **Nearly 2x speedup for Int *, FP +, FP ***
  - Reason: Breaks sequential dependency

```
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
  - Should be (N/2+1)*D cycles: ***cycle per OP ≈ D/2***
  - Measured is slightly worse for FP

Loop Unrolling with Separate Accumulators

```
void unroll2_sa(vec_ptr 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**

Effect of Separate Accumulators

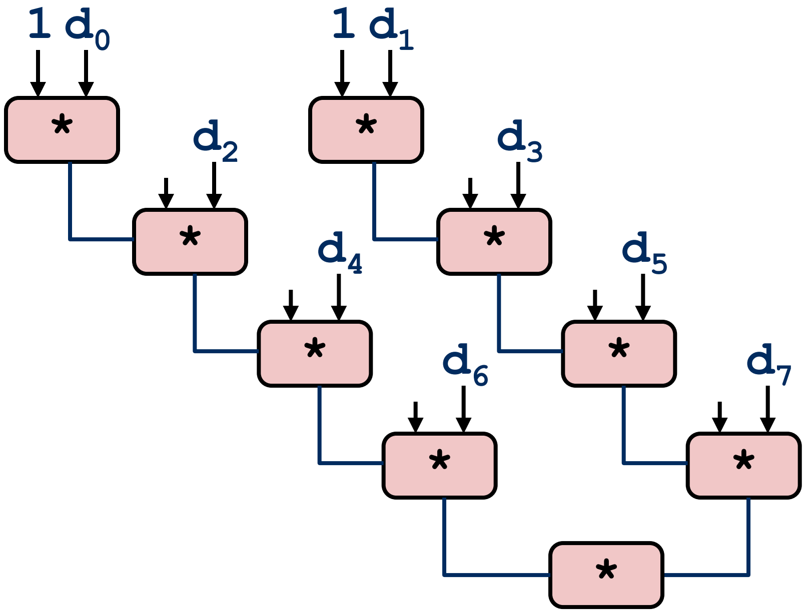
Method	Int (add/mult)		Float (add/mult)	
combine4	2.2	10.0	5.0	7.0
unroll2	1.5	10.0	5.0	7.0
unroll2-ra	1.56	5.0	2.75	3.62
unroll2-sa	1.50	5.0	2.5	3.5
bound	1.0	1.0	2.0	2.0

- **Almost exact 2x speedup (over unroll2) for Int *, FP +, FP ***
  - Breaks sequential dependency in a “cleaner,” more obvious way

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

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:
    cycles per OP ≈ D/2

What Now?

Unrolling & Accumulating

- **Idea**
  - Use K accumulators
  - Increase K until best performance reached
  - Need to unroll by L, K divides L

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

Unrolling & Accumulating: Intel FP *

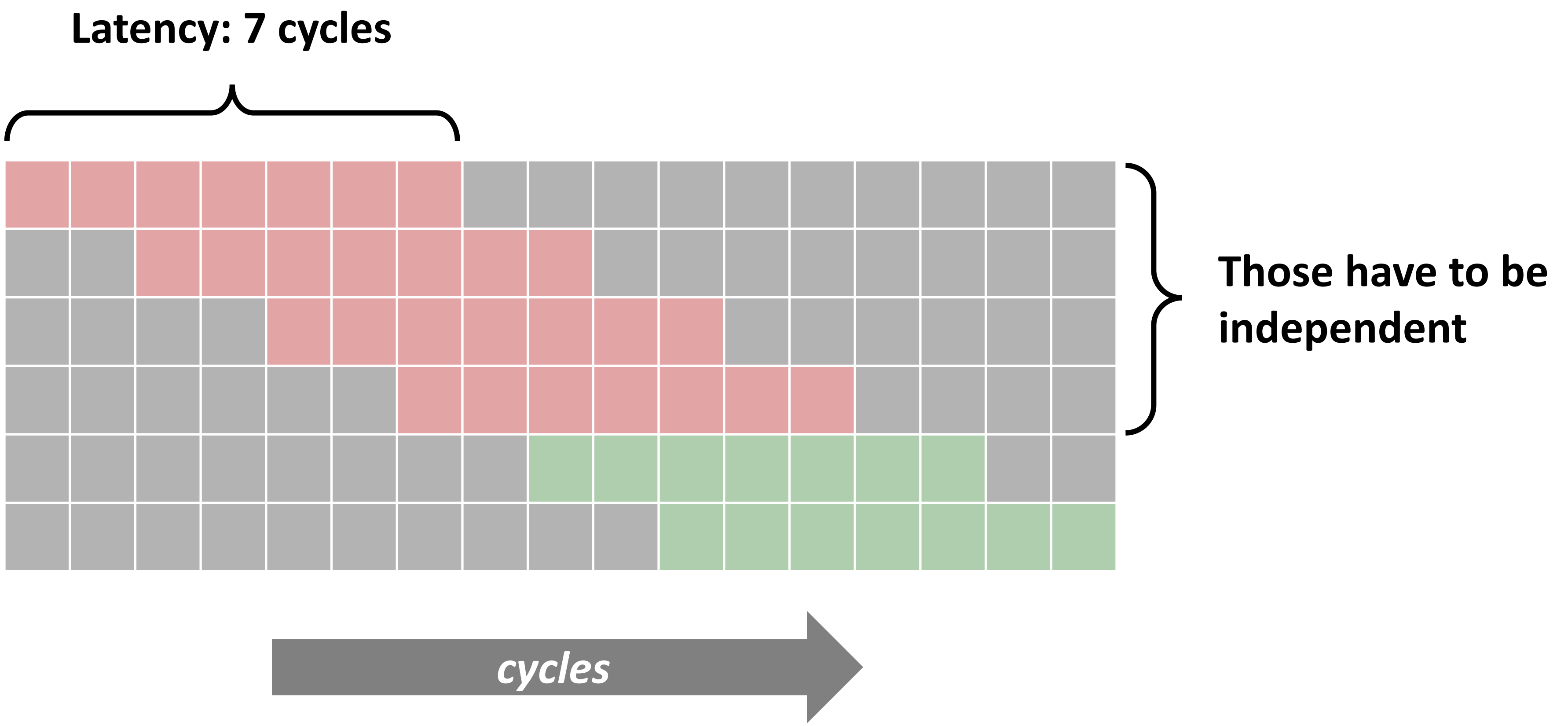
- **Case**
  - Pentium 4
  - FP Multiplication
  - Theoretical Limit: 2.00

Accumulators

FP *	Unrolling Factor L							
K	1	2	3	4	6	8	10	12
1	7.00	7.00		7.01		7.00		
2		3.50		3.50		3.50		
3			2.34					
4				2.01		2.00		
6					2.00			2.01
8						2.01		
10							2.00	
12								2.00

Why 4?

Why 4?



Based on this insight: **K = #accumulators = ceil(latency/cycles per issue)**

Unrolling & Accumulating: Intel FP +

- **Case**
  - Pentium 4
  - FP Addition
  - Theoretical Limit: 2.00

FP +	**Unrolling Factor L**							
K	1	2	3	4	6	8	10	12
1	5.00	5.00		5.02		5.00		
2		2.50		2.51		2.51		
3			2.00					
4				2.01		2.00		
6					2.00			1.99
8						2.01		
10							2.00	
12								2.00

Unrolling & Accumulating: Intel Int *

- **Case**
  - Pentium 4
  - Integer Multiplication
  - Theoretical Limit: 1.00

Int *	Unrolling Factor L							
K	1	2	3	4	6	8	10	12
1	10.00	10.00		10.00		10.01		
2		5.00		5.01		5.00		
3			3.33					
4				2.50		2.51		
6					1.67			1.67
8						1.25		
10							1.09	
12								1.14

Unrolling & Accumulating: Intel Int +

- **Case**
  - Pentium 4
  - Integer addition
  - Theoretical Limit: 1.00

Int +	**Unrolling Factor L**							
K	1	2	3	4	6	8	10	12
1	2.20	1.50		1.10		1.03		
2		1.50		1.10		1.03		
3			1.34					
4				1.09		1.03		
6					1.01			1.01
8						1.03		
10							1.04	
12								1.11

FP *	Unrolling Factor L							
K	1	2	3	4	6	8	10	12
1	7.00	7.00		7.01		7.00		
2		3.50		3.50		3.50		
3			2.34					
4				2.01		2.00		
6					2.00			2.01
8						2.01		
10							2.00	
12								2.00

Pentium 4

FP *	Unrolling Factor L							
K	1	2	3	4	6	8	10	12
1	4.00	4.00		4.00		4.01		
2		2.00		2.00		2.00		
3			1.34					
4				1.00		1.00		
6					1.00			1.00
8						1.00		
10							1.00	
12								1.00

Core 2

*FP * is fully pipelined*

Summary (ILP)

- **Instruction level parallelism may have to be made explicit in program**
- **Potential blockers for compilers**
  - Reassociation changes result (FP)
  - Too many choices, no good way of deciding
- **Unrolling**
  - By itself does often nothing (branch prediction works usually well)
  - But may be needed to enable additional transformations (here: reassociation)
- **How to program this example?**
  - Solution 1: program generator generates alternatives and picks best
  - Solution 2: use model based on latency and throughput

Organization

- Instruction level parallelism (ILP): an example
- **Optimizing compilers and optimization blockers**
  - Overview
  - Removing unnecessary procedure calls
  - Code motion
  - Strength reduction
  - Sharing of common subexpressions
  - Optimization blocker: Procedure calls
  - Optimization blocker: Memory aliasing
  - Summary

Compiler is likely to do that

Optimizing Compilers

-O

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

Example

```
double a[4][4];
double b[4][4];
double c[4][4]; # set to zero

/* Multiply 4 x 4 matrices a and b  */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < 4; i++)
        for (j = 0; j < 4; j++)
            for (k = 0; k < 4; k++)
                c[i*4+j] += a[i*4 + k]*b[k*4 + j];
}
```

Prevents use of SSE

- **Compiled without flags:**
  ~1300 cycles
- **Compiled with -O3 -m64 -march=… -fno-tree-vectorize**
  ~150 cycles
- Core 2 Duo

Optimizing Compilers

- **Compilers are *good* at: mapping program to machine**
  - register allocation
  - code selection and ordering (instruction scheduling)
  - dead code elimination
  - eliminating minor inefficiencies
- **Compilers are *not good* at: algorithmic restructuring**
  - For example to increase ILP, locality, etc.
  - Cannot deal with choices
- **Compilers are *not good* at: overcoming "optimization blockers"**
  - potential memory aliasing
  - potential procedure side-effects

Limitations of Optimizing Compilers

- ***If in doubt, the compiler is conservative***
- **Operate under fundamental constraints**
  - Must not change program behavior under any possible condition
  - Often prevents it from making optimizations when would only affect behavior under pathological conditions
- **Most analysis is performed only within procedures**
  - Whole-program analysis is too expensive in most cases
- **Most analysis is based only on *static* information**
  - Compiler has difficulty anticipating run-time inputs
  - Not good at evaluating or dealing with choices

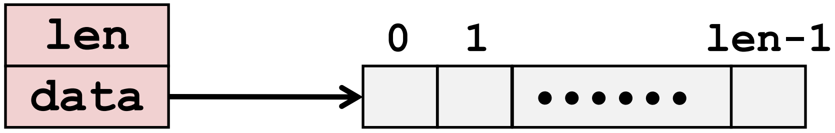
Organization

- Instruction level parallelism (ILP): an example
- **Optimizing compilers and optimization blockers**
  - Overview
  - ***Removing unnecessary procedure calls***
  - Code motion
  - Strength reduction
  - Sharing of common subexpressions
  - Optimization blocker: Procedure calls
  - Optimization blocker: Memory aliasing
  - Summary

Compiler is likely to do that

Example: Data Type for Vectors

```
/* data structure for vectors */
typedef struct{
    int len;
    double *data;
} vec;
```




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

Example: Summing Vector Elements

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

```
/* sum elements of vector */
double sum_elements(vec *v, double *res)
{
  int i;
  n = vec_length(v);
  *res = 0.0;
  double val;

  for (i = 0; i < n; i++) {
    get_vec_element(v, i, &val);
    *res += val;
  }
  return res;
}
```

Overhead for every fp +:

- One fct call
- One <
- One >=
- One ||
- One memory variable access

Slowdown:
probably 10x or more

Removing Procedure Call

```
/* sum elements of vector */
double sum_elements(vec *v, double *res)
{
  int i;
  n = vec_length(v);
  *res = 0.0;
  double val;

  for (i = 0; i < n; i++) {
    get_vec_element(v, i, &val);
     *res += val;
  }
  return res;
}
```

```
/* sum elements of vector */
double sum_elements(vec *v, double *res)
{
  int i;
  n = vec_length(v);
  *res = 0.0;
  double *data = get_vec_start(v);

  for (i = 0; i < n; i++)
     *res += data[i];
  return res;
}
```

Removing Procedure Calls

- **Procedure calls can be very expensive**
- **Bound checking can be very expensive**
- **Abstract data types can easily lead to inefficiencies**
  - Usually avoided for in superfast numerical library functions

- ***Watch your innermost loop!***

- ***Get a feel for overhead versus actual computation being performed***

Organization

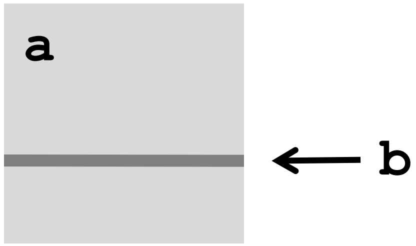
- Instruction level parallelism (ILP): an example
- **Optimizing compilers and optimization blockers**
  - Overview
  - Removing unnecessary procedure calls
  - ***Code motion***
  - Strength reduction
  - Sharing of common subexpressions
  - Optimization blocker: Procedure calls
  - Optimization blocker: Memory aliasing
  - Summary

Compiler is likely to do that

Code Motion

- **Reduce frequency with which computation is performed**
  - If it will always produce same result
  - Especially moving code out of loop (loop-invariant code motion)
- **Sometimes also called precomputation**

```
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];
```

Organization

- Instruction level parallelism (ILP): an example
- **Optimizing compilers and optimization blockers**
  - Overview
  - Removing unnecessary procedure calls
  - Code motion
  - ***Strength reduction***
  - Sharing of common subexpressions
  - Optimization blocker: Procedure calls
  - Optimization blocker: Memory aliasing
  - Summary

Compiler is likely to do that

Strength Reduction

- **Replace costly operation with simpler one**
- **Example: Shift/add instead of multiply or divide**

  `16*x → x << 4`

  - Utility machine dependent
- **Example: 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;
}
```

Organization

- Instruction level parallelism (ILP): an example
- **Optimizing compilers and optimization blockers**
  - Overview
  - Removing unnecessary procedure calls
  - Code motion
  - Strength reduction
  - ***Sharing of common subexpressions***
  - Optimization blocker: Procedure calls
  - Optimization blocker: Memory aliasing
  - Summary

Compiler is likely to do that

Share Common Subexpressions

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

*3 mults: i*n, (i–1)*n, (i+1)*n*

```
/* 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;
```

*1 mult: i*n*

```
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;
```

Organization

- Instruction level parallelism (ILP): an example
- **Optimizing compilers and optimization blockers**
  - Overview
  - Removing unnecessary procedure calls
  - Code motion
  - Strength reduction
  - Sharing of common subexpressions
  - ***Optimization blocker: Procedure calls***
  - Optimization blocker: Memory aliasing
  - Summary

Compiler is likely to do that