How to Write Fast Numerical Code

Spring 2012

Lecture 4

Instructor: Markus Püschel

TA: Georg Ofenbeck

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Technicalities

- **Research project: Let me know**
  - if you know with whom you will work
  - if you have already a project idea
  - current status: on the web
  - Deadline: March 7th

Last Time

- Architecture/Microarchitecture
- Bounds from op count and data loads
- Peak performance of processor
- Performance of code

MMX:
Multimedia extension

SSE:
Streaming SIMD extension

AVX:
Advanced vector extensions

Intel x86	Processors
x86-16	8086
	286
x86-32	386
	486
	Pentium
MMX	Pentium MMX
SSE	Pentium III
SSE2	Pentium 4
SSE3	Pentium 4E
x86-64 / em64t	Pentium 4F
	Core 2 Duo
SSE4	*Penryn*
	Core i7 *(Nehalem)*
AVX	*Sandybridge*

time ↓

Abstracted Microarchitecture: Example Core

Throughput (tp) is measured in doubles/cycle. For example: 2 *(4)*

Latency (lat) is measured in cycles

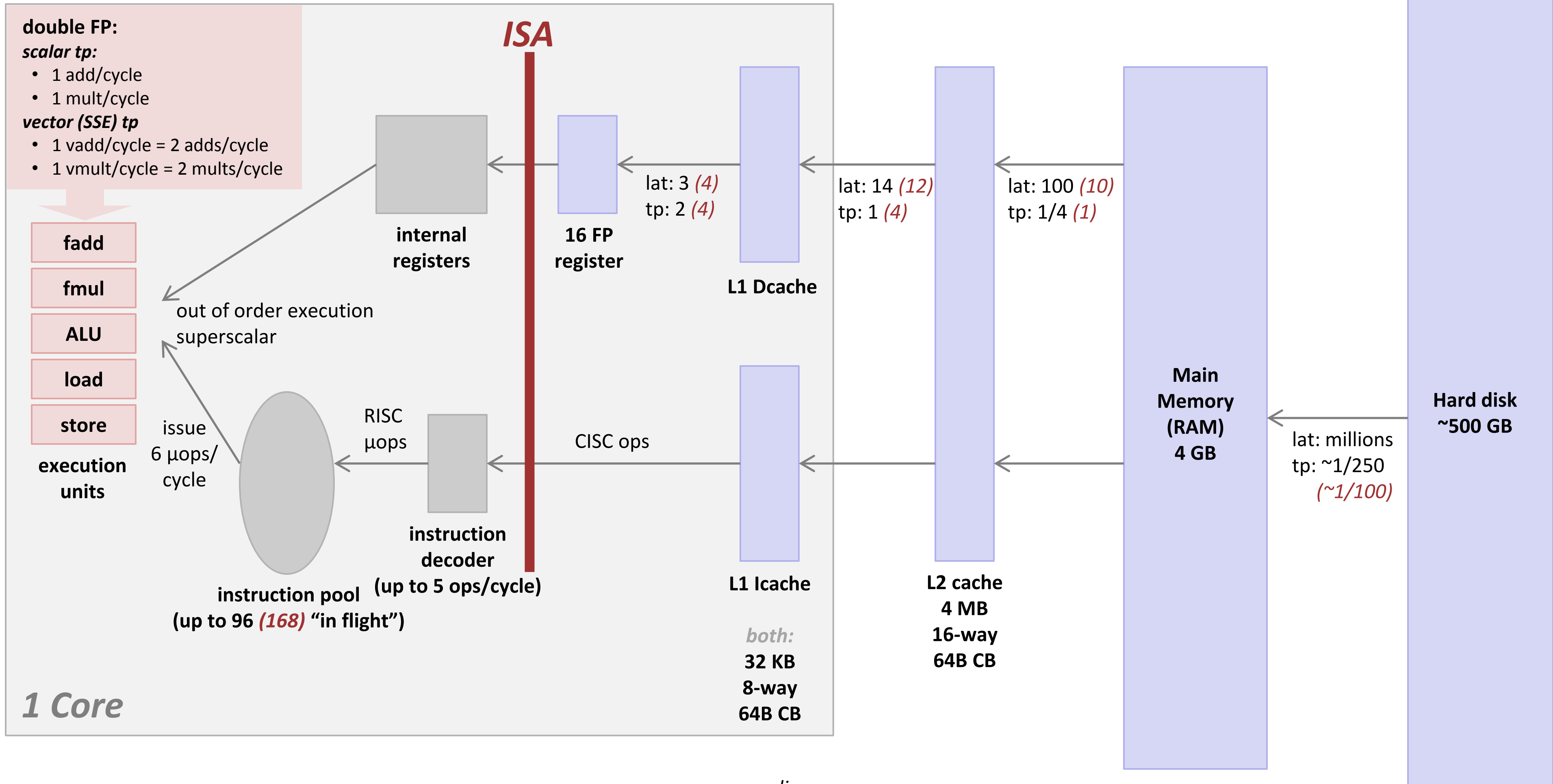
1 double floating point (FP) = 8 bytes

Rectangles not to scale

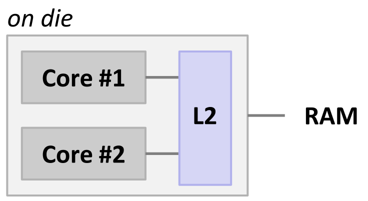
Core 2 (2008) — *Core i7 Sandy Bridge (2011)*

Memory hierarchy:

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



Core 2 Duo:



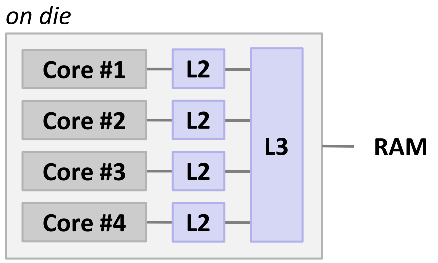
Core i7 Sandy Bridge:

256 KB L2 cache

0.5–2MB L3 cache

vector (AVX) tp

- 1 vadd/cycle = 4 adds/cycle
- 1 vmult/cycle = 4 mults/cycle



How To Make Code Faster?

- **It depends!**
- **Memory bound: Reduce memory traffic**
  - Reduce cache misses, register spills
  - Compress data
- **Compute bound: Keep floating point units busy**
  - Reduce cache misses, register spills
  - Instruction level parallelism (ILP)
  - Vectorization
- **Next: Optimizing for ILP (an example)**

*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

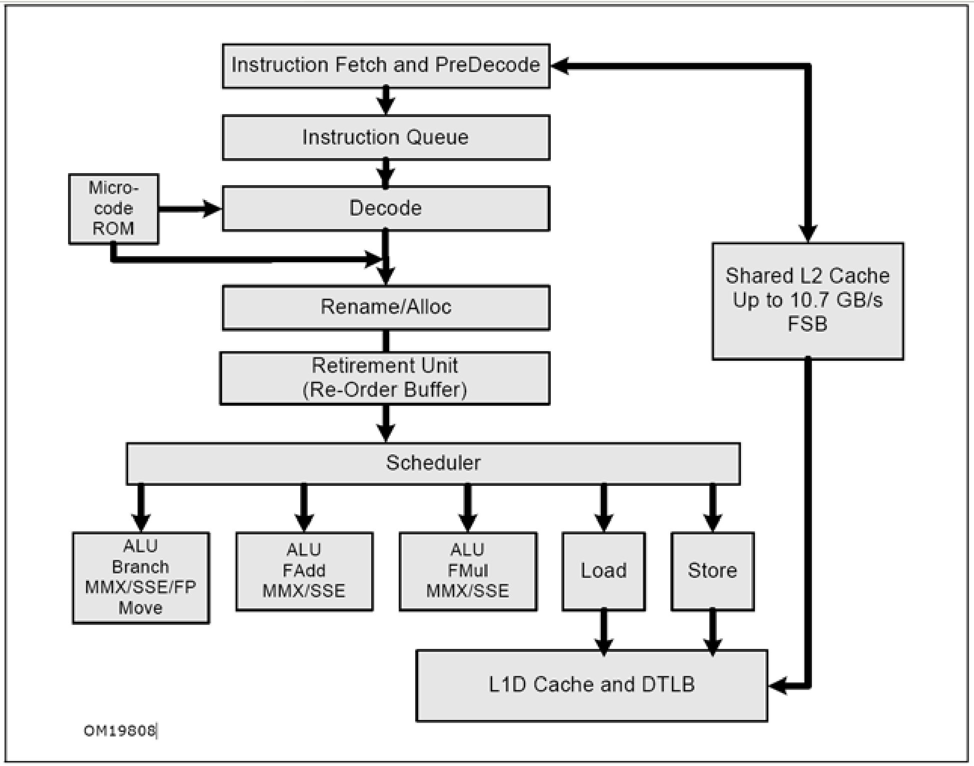

Instruction Fetch and PreDecode
Instruction Queue
Micro-code ROM
Decode
Rename/Alloc
Retirement Unit (Re-Order Buffer)
Scheduler
ALU Branch MMX/SSE/FP Move
ALU FAdd MMX/SSE
ALU FMul MMX/SSE
Load
Store
L1D Cache and DTLB
Shared L2 Cache Up to 10.7 GB/s FSB
OM19808

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: Superscalar processors offer *instruction level parallelism* that can take advantage of fine-grain parallelism many 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?

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**

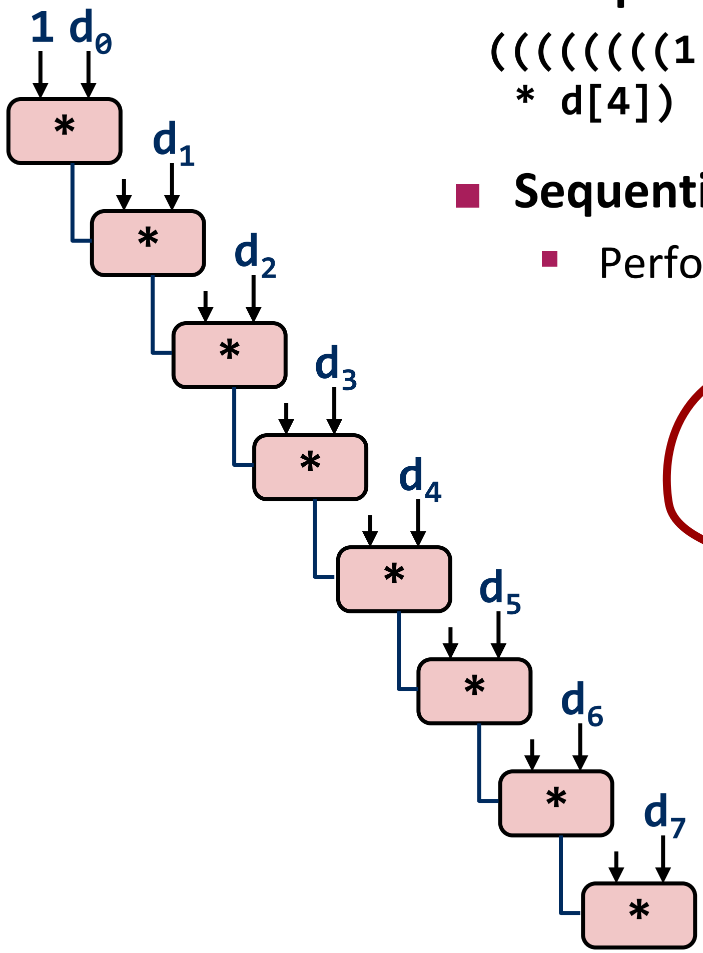
```
void combine4(vec_ptr v, data_t *dest)
{
  int i;
  int length = vec_length(v);
  data_t *d  = get_vec_start(v);
  data_t t   = IDENT;
  for (i = 0; i < length; i++)
    t = t OP d[i];
  *dest = t;
}
```

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

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

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

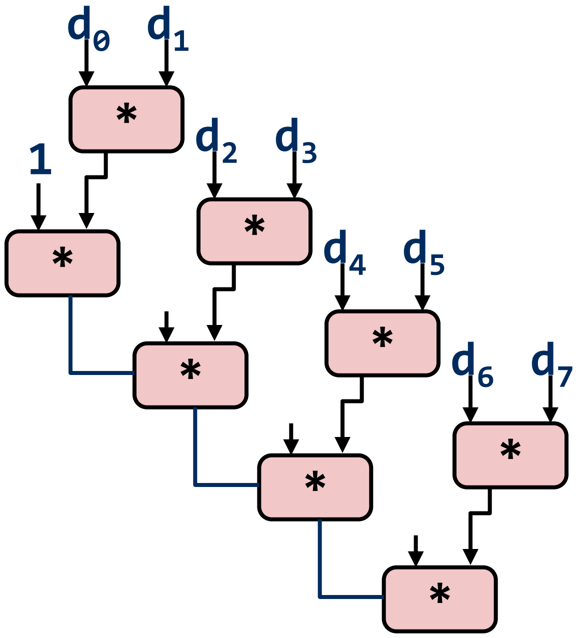
- **Nearly 2x speedup for Int *, FP +, FP ***
  - Reason: Breaks sequential dependency

```
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

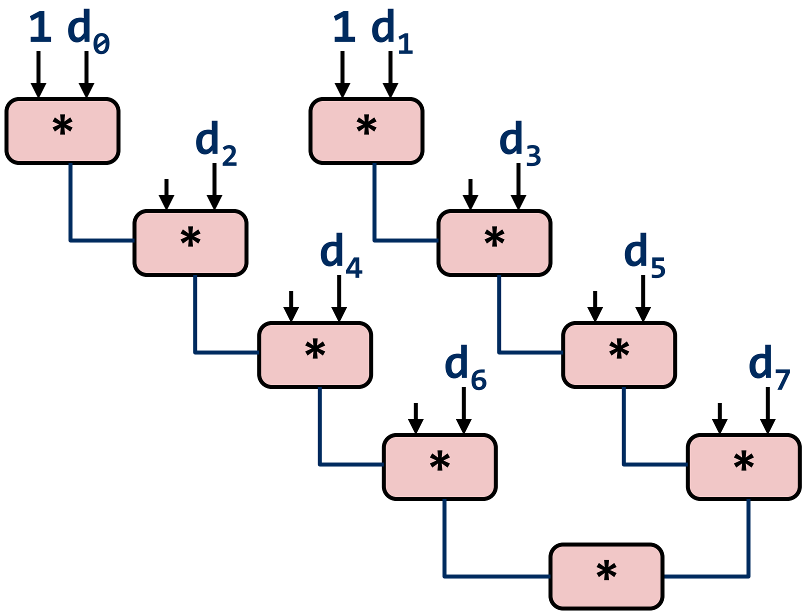
Method	Int (add/mult)		Float (add/mult)	
combine4	2.2	10.0	5.0	7.0
unroll2	1.5	10.0	5.0	7.0
unroll2-ra	1.56	5.0	2.75	3.62
unroll2-sa	1.50	5.0	2.5	3.5
bound	1.0	1.0	2.0	2.0

- **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 *

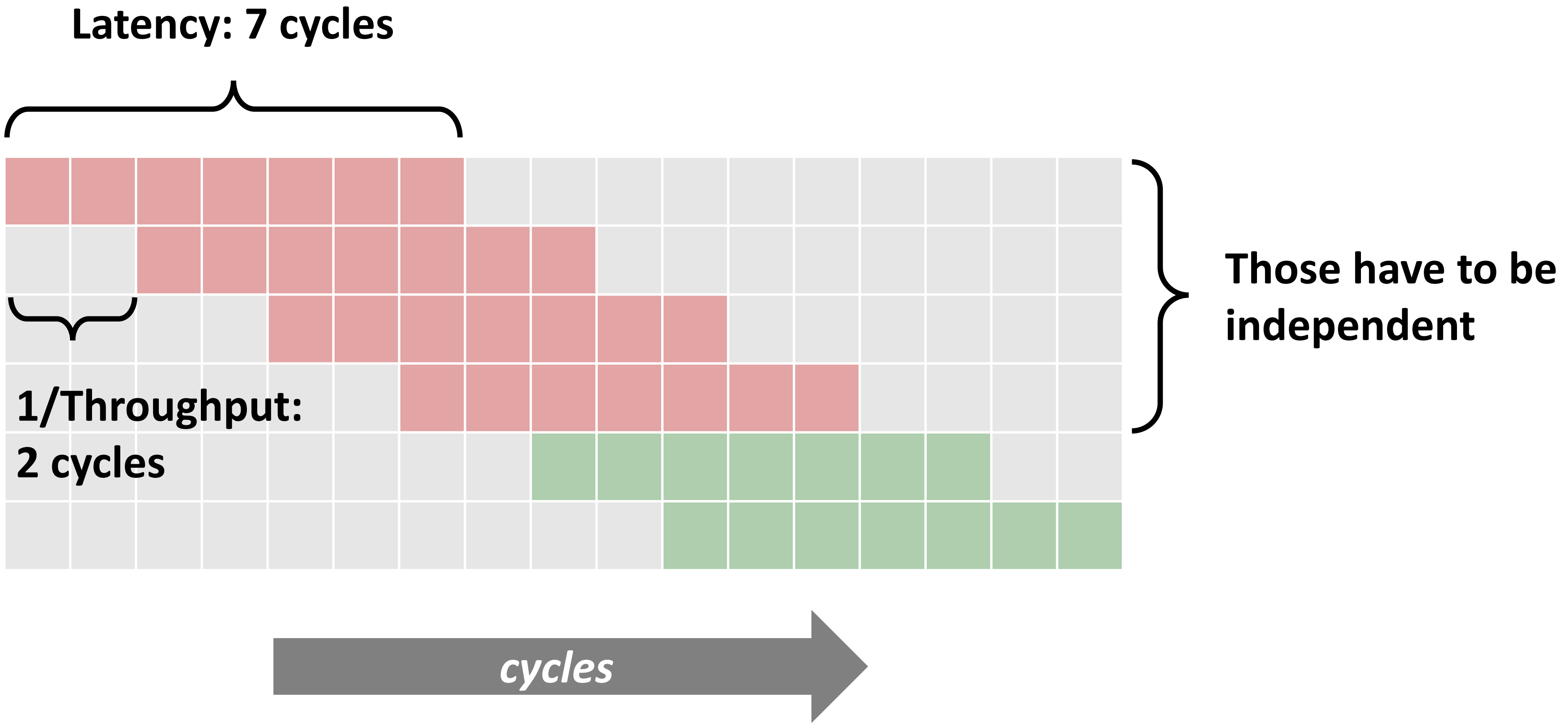
- **Case**
  - Pentium 4
  - FP Multiplication
  - Theoretical Limit: 2.00

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