Advanced Systems Lab

Spring 2020

Lecture: Optimization for Instruction-Level Parallelism

Instructor: Markus Püschel, Ce Zhang

TA: Joao Rivera, Bojan Karlas, several more

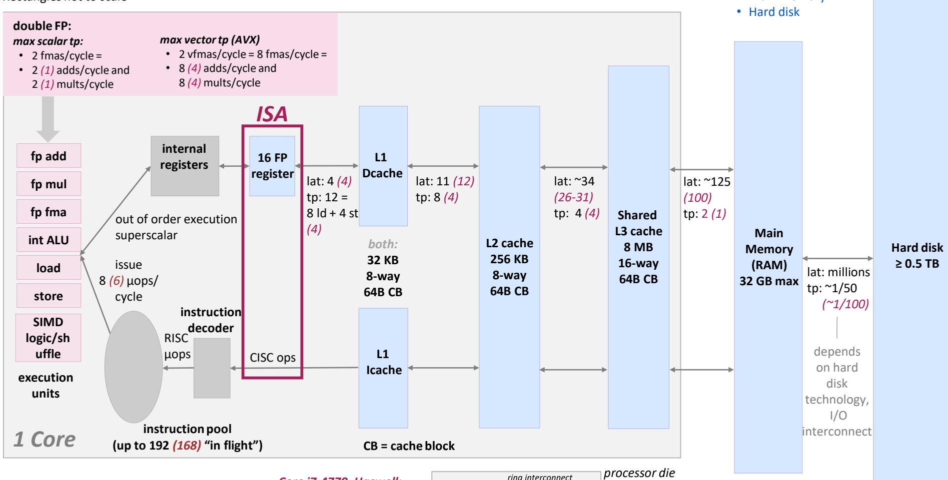
ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Abstracted Microarchitecture: Example Core i7 Haswell (2013) and *Sandybridge (2011)*

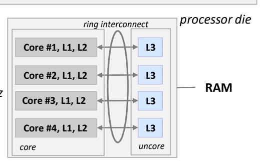
Throughput (tp) is measured in doubles/cycle. For example: 4 *(2)*. *Numbers are for loading into registers.*
Latency (lat) is measured in cycles
1 double floating point (FP) = 8 bytes
fma = fused multiply-add
Rectangles not to scale

Memory hierarchy:

- Registers
- L1 cache
- L2 cache
- L3 cache
- Main memory
- Hard disk

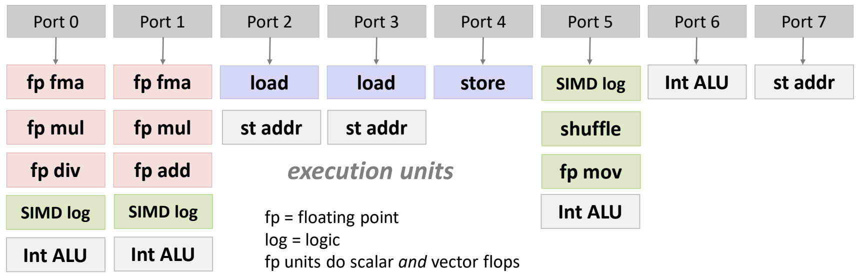


Core i7-4770 Haswell:
4 cores, 8 threads
3.4 GHz
(3.9 GHz max turbo freq)
2 DDR3 channels 1600 MHz



Source: Intel manual (chapter 2), http://www.7-cpu.com/cpu/Haswell.html

Mapping of execution units to ports



fp = floating point
log = logic
fp units do scalar *and* vector flops
SIMD log: other, non-fp SIMD ops

Execution Unit (fp)	Latency [cycles]	Throughput [ops/cycle]	Gap [cycles/issue]
fma	5	2	0.5
mul	5	2	0.5
add	3	1	1
div (scalar) div (4-way)	14-20 25-35	1/13 1/27	13 27

- Every port can issue one instruction/cycle
- Gap = 1/throughput
- ***Intel calls gap the throughput!***
- Same units for scalar and vector flops
- Same latency/throughput for scalar (one double) and AVX vector (four doubles) flops, except for div

Source: Intel manual (Table C-8. 256-bit AVX Instructions, Table 2-6. Dispatch Port and Execution Stacks of the Haswell Microarchitecture, Figure 2-2. CPU Core Pipeline Functionality of the Haswell Microarchitecture),

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*
Part of these slides are adapted from the course associated with this book

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.

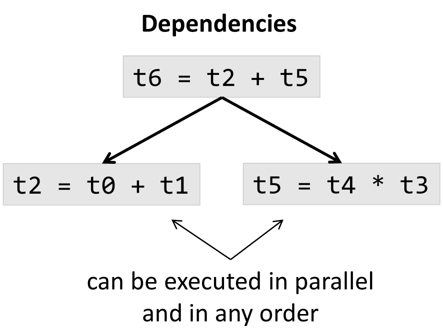
- Benefit: Superscalar processors can take advantage of ***instruction level parallelism (ILP)*** that many programs have

- Most CPUs since about 1998 are superscalar
- Intel: since Pentium Pro
- Simple embedded processors are usually not superscalar

ILP

Code

```
t2 = t0 + t1
t5 = t4 * t3
t6 = t2 + t5
```



Hard Bounds: Haswell and Coffee Lake

Haswell

	latency	1/tp = gap
FP Add	3	1
FP Mul	5	0.5
Int Add	1	0.5
Int Mul	3	1

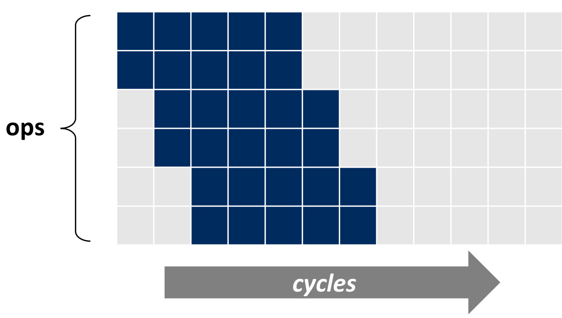
} blackboard

Coffee Lake

	latency	1/tp = gap
FP Add	4	0.5
FP Mul	4	0.5
Int Add	1	0.5
Int Mul	3	1

Haswell

	latency	1/tp = gap
FP Mul	5	0.5



Throughput tp = 2/cycle

Gap = 1/tp = 1/2 cycles/issue

Hard Bounds (cont'd)

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

Example Computation: Reduction

```
void reduce(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_t: double or int

OP: + or *

IDENT: 0 or 1

Runtime of Reduce (Haswell)

- **Use cycles/OP**

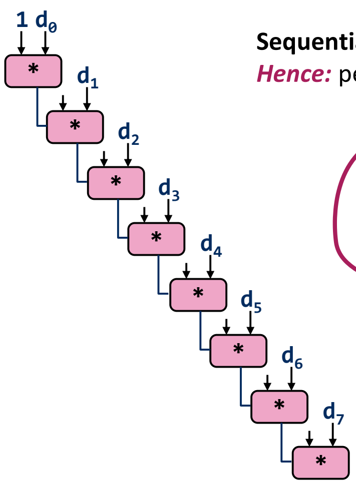
```
void reduce(vec_ptr v, data_t *dest)
{
  int i;
  int length = vec_length(v);
  data_t *d  = get_vec_start(v);
  data_t t   = IDENT;
  for (i = 0; i < length; i++)
    t = t OP d[i];
  *dest = t;
}
```

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

Cycles per OP

Method	Int (add/mult)		Float (add/mult)	
reduce	1.29	2.94	2.95	4.92
bound	0.5	1.0	1.0	0.5

Reduce = Serial Computation (here: *)



Sequential dependence = no ILP!
Hence: performance determined by latency of OP!

Method	Int (add/mult)		Float (add/mult)	
reduce	1.29	2.94	2.95	4.92
bound	0.5	1.0	1.0	0.5

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**
- ***How does the runtime change?***

Effect of Loop Unrolling

Method	Int (add/mult)		Float (add/mult)	
combine4	1.29	2.94	2.95	4.92
unroll2	1.0	2.94	2.95	4.92
bound	0.5	1.0	1.0	0.5

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

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

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;
}
```

- **Can this change the result of the computation?**
- ***Floating point: yes!***

Effect of Separate Accumulators

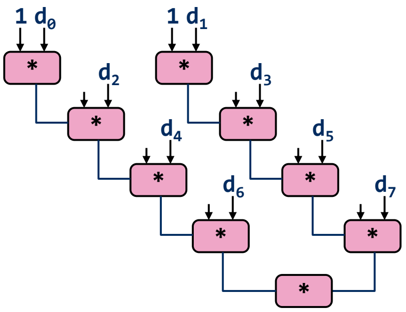
Method	Int (add/mult)		Float (add/mult)	
combine4	1.29	2.94	2.95	4.92
unroll2	1.0	2.94	2.95	4.92
unroll2-sa	0.79	1.49	1.49	2.47
bound	0.5	1.0	1.0	0.5

- **Almost exact 2x speedup (over unroll2) for Int *, FP +, FP ***
  - Breaks sequential dependency

```
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: FP *

- **Haswell: FP multiplication**
  - Gap = cycles/issue = 0.5
  - Latency = 5

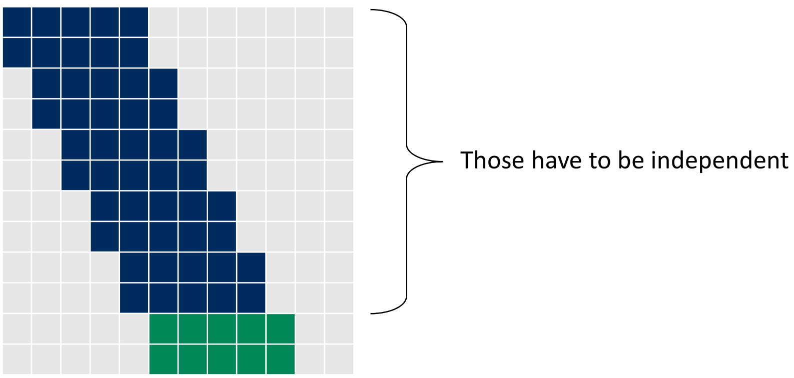
Accumulators

FP64 *	Unrolling Factor L							
K	1	2	3	4	6	8	10	12
1	4.92	4.92		4.92		4.92		
2		2.47		2.47		2.47		
3			1.65		1.65			
4				1.24		1.24		
6					0.85			0.85
8						0.65		
10							0.54	
12								0.52

Why 10?

19

Why 10?



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

Here: **K = ceil(5/0.5) = 10**

20

Unrolling & Accumulating: FP +

- **Haswell: FP addition**
  - Gap = cycles/issue = 1
  - Latency = 3

Accumulators

FP64 +	Unrolling Factor L							
K	1	2	3	4	6	8	10	12
1	2.95	2.95		2.95		2.95		
2		1.49		1.49		1.49		
3			1.00		1.00			
4				1.01		1.01		
6					1.01			1.01
8						1.00		
10							1.01	
12								1.01

Unrolling & Accumulating: Int *

- **Haswell: Int multiplication**
  - Gap = cycles/issue = 1
  - Latency = 3

Accumulators

Int *	Unrolling Factor L							
K	1	2	3	4	6	8	10	12
1	2.94	2.94		2.93		2.93		
2		1.49		1.49		1.49		
3			1.00		1.00			
4				1.00		1.00		
6					1.01			1.00
8						1.01		
10							1.01	
12								1.01

Unrolling & Accumulating: Int +

- **Haswell: Int multiplication**
  - Gap = cycles/issue = 0.5
  - Latency = 1

Accumulators

Int +	Unrolling Factor L							
K	1	2	3	4	6	8	10	12
1	1.29	1.00		1.00		1.00		
2		0.79		0.58		0.53		
3			0.74		0.56			
4				0.58		0.55		
6					0.56			0.53
8						0.53		
10							0.53	
12								0.53

Interesting question: what exactly happens here?

Haswell vs. Coffee Lake: FP +

FP64 +	Unrolling Factor L							
K	1	2	3	4	6	8	10	12
1	2.95	2.95		2.95		2.95		
2		1.49		1.49		1.49		
3			1.00		1.00			
4				1.01		1.01		
6					1.01			1.01
8						1.00		
10							1.01	
12								1.01

Haswell:
Latency = 3
Gap = 1

FP64 +	Unrolling Factor L							
K	1	2	3	4	6	8	10	12
1	3.91	3.90		3.90		3.90		
2		1.96		1.96		1.96		
3			1.32		1.32			
4				1.00		1.00		
6					0.70			0.70
8						0.56		
10							0.54	
12								0.54

Coffee Lake:
Latency = 4
Gap = 0.5

Says something about porting processor-tuned code

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