# Advanced Systems Lab

Spring 2024
*Lecture:* Optimization for Instruction-Level Parallelism
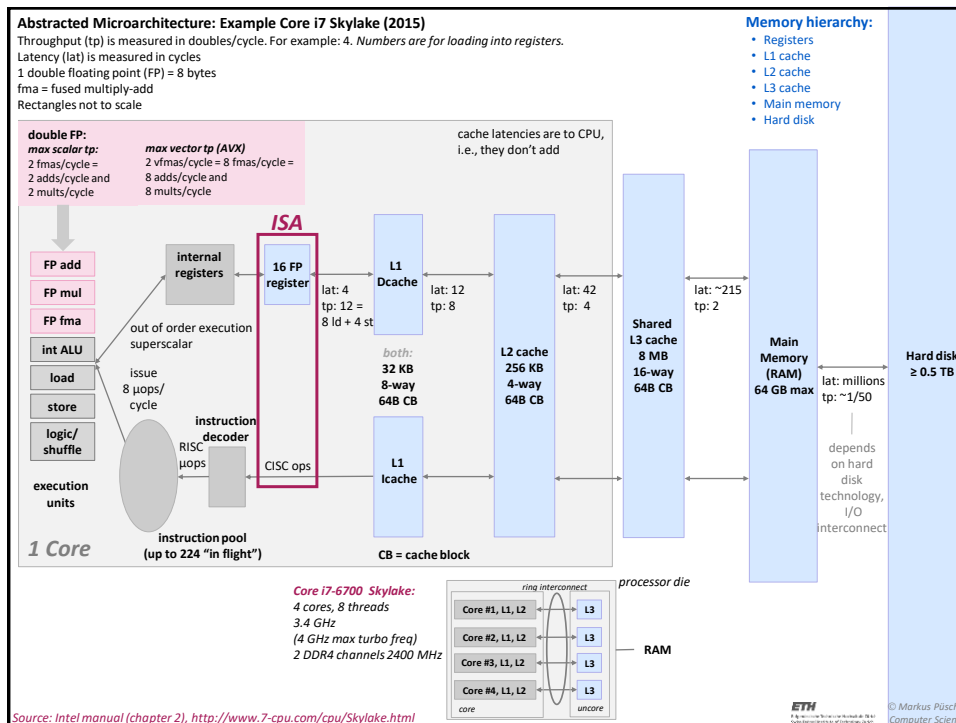
**Instructor:** Markus Püschel

**TA:** Tommaso Pegolotti, several more
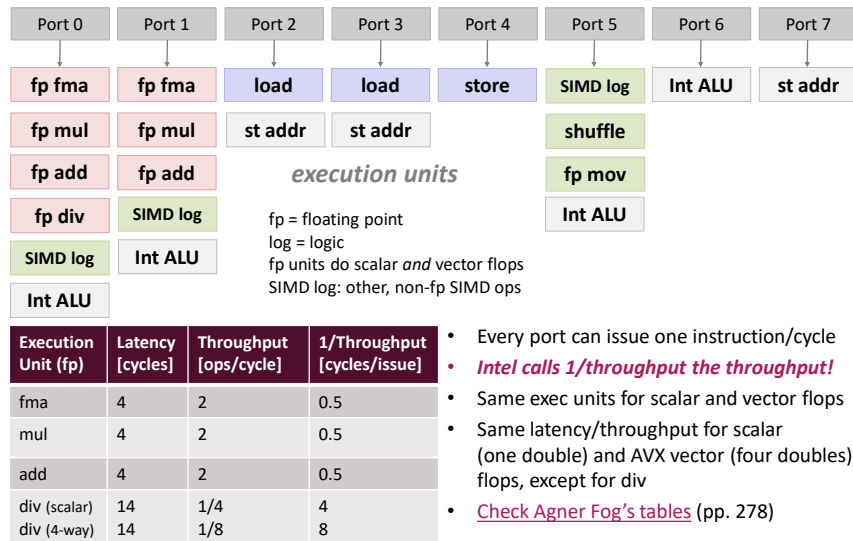
**ETH**
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

1

---

**Abstracted Microarchitecture: Example Core i7 Skylake (2015)**
Throughput (tp) is measured in doubles/cycle. For example: 4. *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



2

# Execution Units and Ports (Skylake)

| Port 0 | Port 1 | Port 2 | Port 3 | Port 4 | Port 5 | Port 6 | Port 7 |
|--------|--------|--------|--------|--------|--------|--------|--------|
| fp fma | fp fma | load | load | store | SIMD log | Int ALU | st addr |
| fp mul | fp mul | st addr | st addr | | shuffle | | |
| fp add | fp add | | | | fp mov | | |
| fp div | SIMD log | | | | Int ALU | | |
| SIMD log | Int ALU | | | | | | |
| Int ALU | | | | | | | |

*execution units*

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] | 1/Throughput [cycles/issue] |
|---------------------|------------------|------------------------|------------------------------|
| fma | 4 | 2 | 0.5 |
| mul | 4 | 2 | 0.5 |
| add | 4 | 2 | 0.5 |
| div (scalar) | 14 | 1/4 | 4 |
| div (4-way) | 14 | 1/8 | 8 |

- Every port can issue one instruction/cycle
- ***Intel calls 1/throughput the throughput!***
- Same exec units for scalar and vector flops
- Same latency/throughput for scalar (one double) and AVX vector (four doubles) flops, except for div
- Check Agner Fog's tables (pp. 278)

*Source: Intel manual (Table C-8. 256-bit AVX Instructions, Table 2-1. Dispatch Port and Execution Stacks of the Skylake Microarchitecture, Figure 2-1. CPU Core Pipeline Functionality of the Skylake Microarchitecture)*

3

---

# How To Make Code Faster?

It depends! First high-level approaches:

Memory bound: Reduce memory traffic
- *Reduce cache misses*
- *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*

4

4

© *Markus Püschel*
*Computer Science*
ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

*Advanced Systems Lab*
*Spring 2024*

# 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 desktop/laptop CPUs since about 1998 are superscalar

Intel: since Pentium Pro

Simple embedded processors are usually not superscalar

***Deep pipelines also require ILP (explained today).***

5

**5**

# ILP

can be executed in parallel
and in any order

**Code**

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

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

**Dependencies**

```
t6 = t2 + t5
```

6

**6**

# Hard Bounds: Coffee Lake and Haswell

**Coffee Lake (2017, Skylake family)**

|        | latency | 1/throughput |
|--------|---------|--------------|
| FP Add | 4       | 0.5          |
| FP Mul | 4       | 0.5          |
| Int Add | 1      | 0.5          |
| Int Mul | 3      | 1            |

**blackboard**

**Haswell (2013)**

|        | latency | 1/throughput |
|--------|---------|--------------|
| FP Add | 3       | 1            |
| FP Mul | 5       | 0.5          |
| Int Add | 1      | 0.5          |
| Int Mul | 3      | 1            |

More precisely (Int Add):
- tp = 2 if one operand is from mem
- tp = 4 if all operands are in register

Only the ALU at Port 1 does Int Mults

7

---

**Coffee Lake**

|        | latency | 1/throughput |
|--------|---------|--------------|
| FP Mul | 4       | 0.5          |
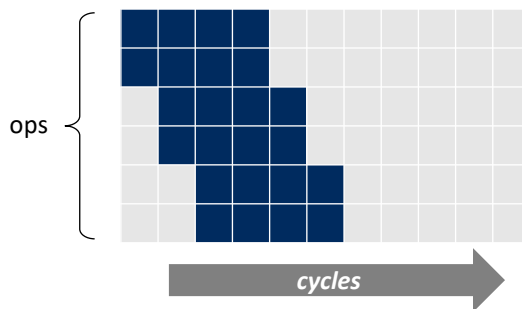


Throughput: 2/cycle

ops

*cycles*

How many cycles at least for n mults?
- *ceil(n/2)*      *(considering only throughput)*
- *ceil(n/2) + 3  (considering latency and throughput)*

8

© *Markus Püschel*
*Computer Science*   ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

*Advanced Systems Lab*
*Spring 2024*

## Example Computation: Reduction

```c
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 (Coffee Lake)

```c
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;
}
```

Measured cycles per OP

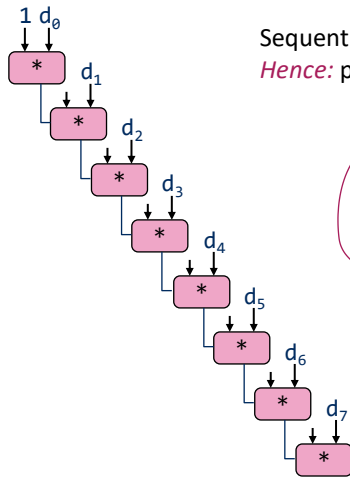| Method | Int (add/mult) | | Float (add/mult) | |
|--------|------|------|------|------|
| reduce | 1.29 | 2.95 | 3.91 | 3.91 |
| bound  | 0.5  | 1.0  | 0.5  | 0.5  |

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

This and all following measurements: gcc -O3 -mavx2 -fno-tree-vectorize

© *Markus Püschel*
*Computer Science*   ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

*Advanced Systems Lab*
*Spring 2024*

# 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 | 2.95 | 3.91 | 3.91 |
| bound  | 0.5  | 1.0  | 0.5  | 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 | 2.95 | 3.91 | 3.91 |
| unroll2 | 1.0 | 2.93 | 3.90 | 3.91 |
| bound | 0.5 | 1.0 | 0.5 | 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;
}
```

*Effect on runtime?*

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 | 2.95 | 3.91 | 3.91 |
| unroll2 | 1.0 | 2.93 | 3.90 | 3.91 |
| unroll2-sa | 0.8 | 1.49 | 1.96 | 1.97 |
| bound | 0.5 | 1.0 | 0.5 | 0.5 |

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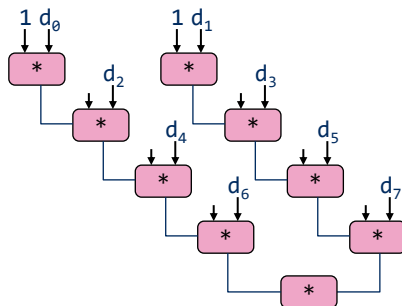
- *Breaks sequential dependency*

```
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*
- *Some overhead for short lengths: Finish off iterations sequentially*

17

**17**

---

# Unrolling & Accumulating: FP *

Coffee Lake: FP multiplication
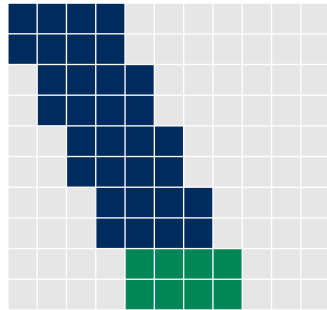- *1/Throughput = cycles/issue = 0.5*
- *Throughput = 2*
- *Latency = 4*

| FP64 * | Unrolling Factor L | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| K | 1 | 2 | 3 | 4 | 6 | 8 | 10 | 12 |
| 1 | 3.91 | 3.91 | | 3.91 | | 3.91 | | |
| 2 | | 1.97 | | 1.97 | | 1.96 | | |
| 3 | | | 1.32 | | 1.32 | | | |
| 4 | | | | 1.00 | | 1.0 | | |
| 6 | | | | | 0.70 | | | 0.70 |
| 8 | | | | | | 0.56 | | |
| 10 | | | | | | | 0.54 | |
| 12 | | | | | | | | 0.54 |

*Accumulators*

*Why 8?*

18

**18**

# Why 8?



Those have to be independent

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

Here (FP Mult):           K = ceil(4 / 0.5) = ceil(4 * 2) = 8

# Unrolling & Accumulating: FP +

Coffee Lake: FP addition

- *Throughput = 2*
- *Latency = 4*

| 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 |

*Accumulators*

# Unrolling & Accumulating: Int *

Coffee Lake: Int multiplication

- *Throughput = 1*
- *Latency = 3*

| 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 | | | 1.32 | | 1.32 | | | |
| 4 | | | | 1.01 | | 1.01 | | |
| 6 | | | | | 1.01 | | | 1.00 |
| 8 | | | | | | 1.01 | | |
| 10 | | | | | | | 1.01 | |
| 12 | | | | | | | | 1.01 |

21

# Unrolling & Accumulating: Int +

Coffee Lake: Int multiplication

- *Throughput = 2*
- *Latency = 1*

| Int + | Unrolling Factor L | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| K | 1 | 2 | 3 | 4 | 6 | 8 | 10 | 12 |
| 1 | 1.29 | 1.00 | | 1.00 | | 1.00 | | |
| 2 | | 0.80 | | 0.58 | | 0.52 | | |
| 3 | | | 0.69 | | 0.52 | | | |
| 4 | | | | 0.57 | | 0.52 | | |
| 6 | | | | | 0.52 | | | 0.52 |
| 8 | | | | | | 0.52 | | |
| 10 | | | | | | | 0.52 | |
| 12 | | | | | | | | 0.52 |

*Interesting question: what exactly happens here?*

22

# Coffee Lake vs. Haswell: FP +

| 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
Throughput = 2

| 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
Throughput = 1

*Says something about porting processor-tuned code*

23

**23**

# Apple M1: FP +

| FP64 + | Unrolling Factor L | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| K | 1 | 2 | 3 | 4 | 6 | 8 | 10 | 12 |
| 1 | 2.87 | 2.83 | | 2.75 | | 2.75 | | |
| 2 | | 1.38 | | 1.39 | | 1.26 | | |
| 3 | | | 0.93 | | 0.93 | | | |
| 4 | | | | 0.71 | | 0.71 | | |
| 6 | | | | | 0.49 | | | 0.49 |
| 8 | | | | | | 0.38 | | |
| 10 | | | | | | | 0.32 | |
| 12 | | | | | | | | 0.31 |

*Firestorm:*
Latency = 3
Throughput = 4

*about 9x speedup!*

24

**24**

# Summary (ILP)

Deep pipelines and multiple ports require ILP for good performance

ILP may have to be made explicit in program

Potential blockers for compilers
- *Reassociation changes result (floating point)*
- *Too many choices, no good way of deciding*

Unrolling
- *By itself does usually 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*

We have seen how to obtain runtime bounds based on throughput and runtime
estimates considering latency. Can be done on arbitrary computation DAGs.  25

**25**