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

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

Latency (lat) is measured in cycles, where 1 double floating point (FP) = 8 bytes.

fma = fused multiply add

Rectangles not to scale

Hard disk ≥ 0.5 TB

Memory hierarchy:
- Registers
- L1 cache
- L2 cache
- L3 cache
- Main memory
- Hard disk

- ISA
- Instruction pool (up to 192 (168) "in flight")
- Instruction decoder
- Issue queue
- Out of order execution superscalar
- Internal registers
- FP add
- FP mul
- FP fma
- int ALU
- load
- store
- Logic/shuffle
- Execution units

1 Core

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

Mapping of execution units to ports

<table>
<thead>
<tr>
<th>Port 0</th>
<th>Port 1</th>
<th>Port 2</th>
<th>Port 3</th>
<th>Port 4</th>
<th>Port 5</th>
<th>Port 6</th>
<th>Port 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>fp fma</td>
<td>fp fma</td>
<td>load</td>
<td>load</td>
<td>store</td>
<td>SIMD log</td>
<td>Int ALU</td>
<td>st addr</td>
</tr>
<tr>
<td>fp mul</td>
<td>fp mul</td>
<td>st addr</td>
<td>st addr</td>
<td>shuffle</td>
<td>fp mov</td>
<td>Int ALU</td>
<td></td>
</tr>
<tr>
<td>fp div</td>
<td>fp add</td>
<td></td>
<td></td>
<td></td>
<td>SIMD log</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SIMD log</td>
<td>SIMD log</td>
<td>Int ALU</td>
<td>Int ALU</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Execution Units (fp)**

- fma
- mul
- add
- div (scalar)
- div (4-way)

**Latency (cycles)**

- fma: 5 cycles
- mul: 5 cycles
- add: 3 cycles
- div (scalar): 14-20 cycles
- div (4-way): 25-35 cycles

**Throughput (ops/cycle)**

- fma: 2 ops/cycle
- mul: 2 ops/cycle
- add: 1 ops/cycle
- div (scalar): 1/13 ops/cycle
- div (4-way): 1/27 ops/cycle

**Gap (cycles/issue)**

- fma: 0.5 cycles
- mul: 0.5 cycles
- add: 1 cycle
- div (scalar): 13 cycles
- div (4-way): 27 cycles

- 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

**How To Make Code Faster?**

It depends!

**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*
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.

Deep pipelines also require ILP (explained today).

Most CPUs since about 1998 are superscalar

Intel: since Pentium Pro

Simple embedded processors are usually not superscalar

ILP

Code
\[
\begin{align*}
t_2 &= t_0 + t_1 \\
t_5 &= t_4 \ast t_3 \\
t_6 &= t_2 + t_5
\end{align*}
\]

Dependencies
\[
\begin{align*}
t_6 &= t_2 + t_5 \\
t_2 &= t_0 + t_1 \\
t_5 &= t_4 \ast t_3
\end{align*}
\]

Can be executed in parallel and in any order
Hard Bounds: Haswell and Coffee Lake

Haswell

<table>
<thead>
<tr>
<th>latency</th>
<th>1/tp = gap</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP Add</td>
<td>3</td>
</tr>
<tr>
<td>FP Mul</td>
<td>5</td>
</tr>
<tr>
<td>Int Add</td>
<td>1</td>
</tr>
<tr>
<td>Int Mul</td>
<td>3</td>
</tr>
</tbody>
</table>

Coffee Lake

<table>
<thead>
<tr>
<th>latency</th>
<th>1/tp = gap</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP Add</td>
<td>4</td>
</tr>
<tr>
<td>FP Mul</td>
<td>4</td>
</tr>
<tr>
<td>Int Add</td>
<td>1</td>
</tr>
<tr>
<td>Int Mul</td>
<td>3</td>
</tr>
</tbody>
</table>

How many cycles at least for n mults?
- \( \text{ceil}(n/2) + 4 \text{(considering latency and throughput)} \)
- \( \text{ceil}(n/2) \text{(considering only throughput)} \)

Throughput \( tp = 2/\text{cycle} \)

Gap = \( 1/tp = 1/2 \text{ cycles/issue} \)
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;
}
```


data_t: double or int

OP: + or *
IDENT: 0 or 1

Runtime of Reduce (Haswell)

```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

<table>
<thead>
<tr>
<th>Method</th>
<th>Int (add/mult)</th>
<th>Float (add/mult)</th>
</tr>
</thead>
<tbody>
<tr>
<td>reduce</td>
<td>1.29 2.94 2.95</td>
<td>4.92</td>
</tr>
<tr>
<td>bound</td>
<td>0.5 1.0 1.0</td>
<td>0.5</td>
</tr>
</tbody>
</table>

Questions:
- Explain red row
- Explain gray row

This and all following measurements: gcc -O3 -mavx2 -fno-tree-vectorize
Reduce = Serial Computation (here: *)

Sequential dependence = no ILP!

Hence: performance determined by latency of OP!

<table>
<thead>
<tr>
<th>Method</th>
<th>Int (add/mult)</th>
<th>Float (add/mult)</th>
</tr>
</thead>
<tbody>
<tr>
<td>reduce</td>
<td>1.29 2.94 2.95 4.92</td>
<td></td>
</tr>
<tr>
<td>bound</td>
<td>0.5 1.0 1.0 0.5</td>
<td></td>
</tr>
</tbody>
</table>

Loop Unrolling

```c
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**

<table>
<thead>
<tr>
<th>Method</th>
<th>Int (add/mult)</th>
<th>Float (add/mult)</th>
</tr>
</thead>
<tbody>
<tr>
<td>combine4</td>
<td>1.29</td>
<td>2.94</td>
</tr>
<tr>
<td></td>
<td>2.95</td>
<td>4.92</td>
</tr>
<tr>
<td>unroll2</td>
<td>1.0</td>
<td>2.94</td>
</tr>
<tr>
<td></td>
<td>2.95</td>
<td>4.92</td>
</tr>
<tr>
<td>bound</td>
<td>0.5</td>
<td>1.0</td>
</tr>
<tr>
<td></td>
<td>1.0</td>
<td>0.5</td>
</tr>
</tbody>
</table>

Helps integer sum a bit

Others don’t improve. Why?
- Still sequential dependency

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

---

**Loop Unrolling with Separate Accumulators**

```c
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!*

---

Adv. Systems Lab
Spring 2021

© Markus Püschel
Computer Science
ETH Zürich
Swiss Federal Institute of Technology
Zürich
Effect of Separate Accumulators

<table>
<thead>
<tr>
<th>Method</th>
<th>Int (add/mult)</th>
<th>Float (add/mult)</th>
</tr>
</thead>
<tbody>
<tr>
<td>combine4</td>
<td>1.29</td>
<td>2.94</td>
</tr>
<tr>
<td></td>
<td>2.95</td>
<td>4.92</td>
</tr>
<tr>
<td>unroll2</td>
<td>1.0</td>
<td>2.94</td>
</tr>
<tr>
<td></td>
<td>2.95</td>
<td>4.92</td>
</tr>
<tr>
<td>unroll2-sa</td>
<td>0.79</td>
<td>1.49</td>
</tr>
<tr>
<td></td>
<td>1.49</td>
<td>2.47</td>
</tr>
<tr>
<td>bound</td>
<td>0.5</td>
<td>1.0</td>
</tr>
<tr>
<td></td>
<td>1.0</td>
<td>0.5</td>
</tr>
</tbody>
</table>

Almost exact 2x speedup (over unroll2) for Int *, FP +, FP *

- Breaks sequential dependency

\[ x_0 = x_0 \text{ OP } d[i]; \]
\[ x_1 = x_1 \text{ OP } d[i+1]; \]

Separate Accumulators

What changed:
- Two independent "streams" of operations

Overall Performance
- N elements, D cycles latency/op
- Should be \((N/2+1)\times D\) cycles:
  \(\text{cycles per OP} = D/2\)

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:
  \( \text{Cannot go beyond throughput limitations of execution units} \)
- Large overhead for short lengths: Finish off iterations sequentially

Unrolling & Accumulating: FP *

Haswell: FP multiplication

- \( \text{Gap} = \text{cycles/issue} = 0.5 \)
- \( \text{Latency} = 5 \)

<table>
<thead>
<tr>
<th>FP64 *</th>
<th>Unrolling Factor L</th>
</tr>
</thead>
<tbody>
<tr>
<td>( K )</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>4.92</td>
</tr>
<tr>
<td>2</td>
<td>2.47</td>
</tr>
<tr>
<td>3</td>
<td>1.65</td>
</tr>
<tr>
<td>4</td>
<td>1.24</td>
</tr>
<tr>
<td>6</td>
<td>0.85</td>
</tr>
<tr>
<td>8</td>
<td>0.65</td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
</tr>
</tbody>
</table>

Why 10?
Why 10?

Based on this insight:  
\[ K = \#\text{accumulators} = \text{ceil}(\text{latency/cycles per issue}) \]
\[ = \text{ceil}(\text{latency} \times \text{throughput}) \]

Here:  
\[ K = \text{ceil}(5/0.5) = 10 \]

Unrolling & Accumulating: FP +

Haswell: FP addition

- \textit{Gap} = \text{cycles/issue} = 1
- \textit{Latency} = 3

<table>
<thead>
<tr>
<th>Accumulators</th>
<th>FP64 +</th>
<th>Unrolling Factor L</th>
</tr>
</thead>
<tbody>
<tr>
<td>K 1 2 3 4 6 8 10 12</td>
<td>1 2.95 2.95 2.95</td>
<td>1.01 1.01 1.01 1.01</td>
</tr>
</tbody>
</table>
Unrolling & Accumulating: Int *

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

<table>
<thead>
<tr>
<th>Int *</th>
<th>Unrolling Factor L</th>
</tr>
</thead>
<tbody>
<tr>
<td>K</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2.94</td>
</tr>
<tr>
<td>2</td>
<td>1.49</td>
</tr>
<tr>
<td>3</td>
<td>1.00</td>
</tr>
<tr>
<td>4</td>
<td>1.00</td>
</tr>
<tr>
<td>6</td>
<td>1.01</td>
</tr>
<tr>
<td>8</td>
<td>1.01</td>
</tr>
<tr>
<td>10</td>
<td>1.01</td>
</tr>
<tr>
<td>12</td>
<td>1.01</td>
</tr>
</tbody>
</table>

Unrolling & Accumulating: Int +

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

<table>
<thead>
<tr>
<th>Int +</th>
<th>Unrolling Factor L</th>
</tr>
</thead>
<tbody>
<tr>
<td>K</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1.29</td>
</tr>
<tr>
<td>2</td>
<td>0.79</td>
</tr>
<tr>
<td>3</td>
<td>0.74</td>
</tr>
<tr>
<td>4</td>
<td>0.58</td>
</tr>
<tr>
<td>6</td>
<td>0.56</td>
</tr>
<tr>
<td>8</td>
<td>0.53</td>
</tr>
<tr>
<td>10</td>
<td>0.53</td>
</tr>
<tr>
<td>12</td>
<td>0.53</td>
</tr>
</tbody>
</table>

Interesting question: what exactly happens here?
### Haswell vs. Coffee Lake: FP +

<table>
<thead>
<tr>
<th>Unrolling Factor L</th>
<th>1  2  3  4  6  8  10 12</th>
</tr>
</thead>
<tbody>
<tr>
<td>K 1</td>
<td>2.95 2.95 2.95 2.95</td>
</tr>
<tr>
<td>2</td>
<td>1.49 1.49 1.49 1.49</td>
</tr>
<tr>
<td>3</td>
<td>1.00 1.00 1.00 1.00</td>
</tr>
<tr>
<td>4</td>
<td>1.01 1.01 1.01 1.01</td>
</tr>
<tr>
<td>6</td>
<td>1.00 1.00 1.00 1.00</td>
</tr>
<tr>
<td>8</td>
<td>1.01 1.01 1.01 1.01</td>
</tr>
<tr>
<td>10</td>
<td>1.01 1.01 1.01 1.01</td>
</tr>
<tr>
<td>12</td>
<td>1.01 1.01 1.01 1.01</td>
</tr>
</tbody>
</table>

**Haswell:**
- Latency = 3
- Gap = 1

<table>
<thead>
<tr>
<th>Unrolling Factor L</th>
<th>1  2  3  4  6  8  10 12</th>
</tr>
</thead>
<tbody>
<tr>
<td>K 1</td>
<td>3.91 3.90 3.90 3.90</td>
</tr>
<tr>
<td>2</td>
<td>1.96 1.96 1.96 1.96</td>
</tr>
<tr>
<td>3</td>
<td>1.32 1.32 1.32 1.32</td>
</tr>
<tr>
<td>4</td>
<td>1.00 1.00 1.00 1.00</td>
</tr>
<tr>
<td>6</td>
<td>0.70 0.70 0.70 0.70</td>
</tr>
<tr>
<td>8</td>
<td>0.56 0.56 0.56 0.56</td>
</tr>
<tr>
<td>10</td>
<td>0.54 0.54 0.54 0.54</td>
</tr>
<tr>
<td>12</td>
<td>0.54 0.54 0.54 0.54</td>
</tr>
</tbody>
</table>

**Coffee Lake:**
- Latency = 4
- Gap = 0.5

*Says something about porting processor-tuned code*

### Summary (ILP)

Deep pipelines require ILP for good throughput

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