What Makes Code Run Fast?

- Many operations have non-zero latencies
- Modern machines can issue several operations per cycle
- Execution time is order-dependent (and has been since the 60's)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>load</td>
<td>3</td>
</tr>
<tr>
<td>store</td>
<td>3</td>
</tr>
<tr>
<td>loadI</td>
<td>1</td>
</tr>
<tr>
<td>add</td>
<td>1</td>
</tr>
<tr>
<td>mult</td>
<td>2</td>
</tr>
<tr>
<td>fadd</td>
<td>1</td>
</tr>
<tr>
<td>fmult</td>
<td>2</td>
</tr>
<tr>
<td>shift</td>
<td>1</td>
</tr>
<tr>
<td>branch</td>
<td>0 to 8</td>
</tr>
</tbody>
</table>

- Loads & stores may or may not block on issue
  - Non-blocking ⇒ fill those issue slots
- Branch costs vary with path taken
- Branches typically have delay slots
  - Fill slots with unrelated operations
  - Percolates branch upward
- Scheduler should hide the latencies

Note by Baris Aktemur:
Our slides are adapted from Cooper and Torczon’s slides that they prepared for COMP 412 at Rice.
Example

\[ a \leftarrow a \times 2 \times b \times c \times d \]

**Simple schedule**

<table>
<thead>
<tr>
<th>Start</th>
<th>Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>loadAI r_{FP},@a \Rightarrow r_1</td>
</tr>
<tr>
<td>4</td>
<td>add r_1, r_2 \Rightarrow r_3</td>
</tr>
<tr>
<td>5</td>
<td>loadAI r_{FP},@b \Rightarrow r_2</td>
</tr>
<tr>
<td>8</td>
<td>mult r_1, r_2 \Rightarrow r_3</td>
</tr>
<tr>
<td>10</td>
<td>loadAI r_{FP},@c \Rightarrow r_2</td>
</tr>
<tr>
<td>13</td>
<td>mult r_1, r_2 \Rightarrow r_3</td>
</tr>
<tr>
<td>15</td>
<td>loadAI r_{FP},@d \Rightarrow r_2</td>
</tr>
<tr>
<td>18</td>
<td>mult r_1, r_2 \Rightarrow r_3</td>
</tr>
<tr>
<td>20</td>
<td>storeAI r_1 \Rightarrow r_{FP},@a</td>
</tr>
</tbody>
</table>

**Schedule loads early**

<table>
<thead>
<tr>
<th>Start</th>
<th>Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>loadAI r_{FP},@a \Rightarrow r_1</td>
</tr>
<tr>
<td>2</td>
<td>loadAI r_{FP},@b \Rightarrow r_2</td>
</tr>
<tr>
<td>3</td>
<td>loadAI r_{FP},@c \Rightarrow r_3</td>
</tr>
<tr>
<td>4</td>
<td>add r_1, r_3 \Rightarrow r_4</td>
</tr>
<tr>
<td>5</td>
<td>mult r_1, r_2 \Rightarrow r_3</td>
</tr>
<tr>
<td>6</td>
<td>loadAI r_{FP},@d \Rightarrow r_2</td>
</tr>
<tr>
<td>7</td>
<td>mult r_1, r_2 \Rightarrow r_3</td>
</tr>
<tr>
<td>9</td>
<td>mult r_1, r_2 \Rightarrow r_3</td>
</tr>
<tr>
<td>11</td>
<td>storeAI r_1 \Rightarrow r_{FP},@a</td>
</tr>
</tbody>
</table>

Reordering operations for speed is called *instruction scheduling*

---

**ALU Characteristics**

This data is surprisingly hard to measure accurately

- Value-dependent behavior
- Context-dependent behavior
- Compiler behavior
  - Have seen gcc underallocate & inflate operation costs with memory references (spills)
  - Have seen commercial compiler generate 3 extra ops per divide raising effective cost by 3
- Difficult to reconcile measured reality with the data in the Manuals (e.g. integer divide on Nehalem)

**Intel E5530 operation latencies**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>64 bit integer subtract</td>
<td>1</td>
</tr>
<tr>
<td>64 bit integer multiply</td>
<td>3</td>
</tr>
<tr>
<td>64 bit integer divide</td>
<td>41</td>
</tr>
<tr>
<td>Double precision add</td>
<td>3</td>
</tr>
<tr>
<td>Double precision subtract</td>
<td>3</td>
</tr>
<tr>
<td>Double precision multiply</td>
<td>5</td>
</tr>
<tr>
<td>Double precision divide</td>
<td>22</td>
</tr>
<tr>
<td>Single precision add</td>
<td>3</td>
</tr>
<tr>
<td>Single precision subtract</td>
<td>3</td>
</tr>
<tr>
<td>Single precision multiply</td>
<td>4</td>
</tr>
<tr>
<td>Single precision divide</td>
<td>14</td>
</tr>
</tbody>
</table>

Xeon E5530 uses the Nehalem microarchitecture, as does i7
Instruction Scheduling (Engineer’s View)

The Problem

Given a code fragment for some target machine and the latencies for each individual operation, reorder the operations to minimize execution time.

The Concept

Scheduler

Machine description

slow code → fast code

The Task

- Produce correct code
- Minimize wasted cycles
- Avoid spilling registers
- Operate efficiently

Instruction Scheduling (The Abstract View)

To capture properties of the code, build a precedence graph $G$

- Nodes $n \in G$ are operations with $type(n)$ and $delay(n)$
- An edge $e = (n_1, n_2) \in G$ if & only if $n_2$ uses the result of $n_1$

The Code

```
loadAI rarp, @a \rightarrow r1
add r1, r1 \rightarrow r1
loadAI rarp, @b \rightarrow r2
mult r1, r2 \rightarrow r3
loadAI rarp, @c \rightarrow r3
mult r1, r2 \rightarrow r1
loadAI rarp, @d \rightarrow r2
mult r1, r2 \rightarrow r1
storeAI r1 \rightarrow rarp, @e
```

The Precedence Graph
Instruction Scheduling (Definitions)

A correct schedule \( S \) maps each \( n \in N \) into a non-negative integer representing its cycle number, and

1. \( S(n) \geq 0 \), for all \( n \in N \), obviously.
2. If \( (n_1, n_2) \in E \), \( S(n_1) + \text{delay}(n_1) \leq S(n_2) \)
3. For each type \( t \), there are no more operations of type \( t \) in any cycle than the target machine can issue.

The length of a schedule \( S \), denoted \( L(S) \), is

\[ L(S) = \max_{n \in N} (S(n) + \text{delay}(n)) \]

The goal is to find the shortest possible correct schedule. \( S \) is time-optimal if \( L(S) \leq L(S_i) \), for all other schedules \( S_i \).

A schedule might also be optimal in terms of registers, power, or space....

Instruction Scheduling (What’s so difficult?)

Critical Points

- All operands must be available
- Multiple operations can be ready
- Moving operations can lengthen register lifetimes
- Placing uses near definitions can shorten register lifetimes
- Operands can have multiple predecessors

Together, these issues make scheduling hard (NP-Complete)

Local scheduling is the simple case

- Restricted to straight-line code
- Consistent and predictable latencies
**Instruction Scheduling: The Big Picture**

1. Build a precedence graph, \( P \)
2. Compute a **priority function** over the nodes in \( P \)
3. Use list scheduling to construct a schedule, 1 cycle at a time
   a. Use a queue of operations that are ready
   b. At each cycle
      i. Choose the highest priority ready operation & schedule it
      ii. Update the ready queue

**Local list scheduling**
- The dominant algorithm for thirty years
- A greedy, heuristic, local technique

---

**Local List Scheduling**

\[
\begin{align*}
\text{Cycle} & \leftarrow 1 \\
\text{Ready} & \leftarrow \text{leaves of } P \\
\text{Active} & \leftarrow \emptyset \\
\text{while} \ (\text{Ready} \cup \text{Active} \neq \emptyset) & \\
\text{if} \ (\text{Ready} = \emptyset) & \text{then} \\
\text{remove an op from Ready} & \\
\text{S(op)} & \leftarrow \text{Cycle} \\
\text{Active} & \leftarrow \text{Active} \cup \text{op} \\
\text{Cycle} & \leftarrow \text{Cycle} + 1 \\
\text{for each op} \in \text{Active} & \\
\text{if} \ (\text{S(op)} + \text{delay(op)} \leq \text{Cycle}) & \text{then} \\
\text{remove op from Active} & \\
\text{for each successor } s \text{ of op in } P & \text{if } (s \text{ is ready}) \text{ then} \\
\text{Ready} & \leftarrow \text{Ready} \cup s
\end{align*}
\]
Scheduling Example

1. Build the precedence graph

The Code

```
loadA \( r_{arg, a} \rightarrow r_1 \\
add \( r_1, r_2 \rightarrow r_1 \\
loadA \( r_{arg, b} \rightarrow r_2 \\
mult \( r_1, r_2 \rightarrow r_1 \\
loadA \( r_{arg, c} \rightarrow r_3 \\
mult \( r_1, r_2 \rightarrow r_1 \\
loadA \( r_{arg, d} \rightarrow r_3 \\
mult \( r_1, r_2 \rightarrow r_1 \\
storeA \( r_1 \rightarrow r_{arg, e} \\
```

The Precedence Graph

```
a
\downarrow
b
\downarrow
\cdots
\downarrow
d
c
\downarrow
f
e
\downarrow
g
\downarrow
h
\downarrow
i
```

2. Determine priorities: longest latency-weighted path

The Code

```
load \ 3
store \ 3
loadI \ 1
add \ 1
mult \ 2
fadd \ 1
fmult \ 2
shift \ 1
branch \ 0 \ to \ 8
```

The Precedence Graph

```
a
\downarrow
13
\downarrow
12
\downarrow
10
\downarrow
9
\cdots
\downarrow
1
f
\downarrow
g
\downarrow
h
\downarrow
i
```
Scheduling Example

1. Build the precedence graph
2. Determine priorities: longest latency-weighted path
3. Perform list scheduling

1) a: loadAl r0, @w ⇒ r1
2) c: loadAl r0, @x ⇒ r2
3) e: loadAl r0, @y ⇒ r3
4) b: add r1, r1 ⇒ r1
5) d: mult r1, r2 ⇒ r1
6) g: loadAl r0, @z ⇒ r2
7) f: mult r1, r3 ⇒ r1
8) h: mult r1, r2 ⇒ r1
9) i: storeAl r1 ⇒ r0, @w

Used a new register name

The Code

More List Scheduling

List scheduling breaks down into two distinct classes

- **Forward list scheduling**
  - Start with available operations
  - Work forward in time
  - Ready ⇒ all operands available

- **Backward list scheduling**
  - Start with no successors
  - Work backward in time
  - Ready ⇒ latency covers uses

Variations on list scheduling

- Prioritize critical path(s)
- Schedule last use as soon as possible
- Depth first in precedence graph (minimize registers)
- Breadth first in precedence graph (minimize interlocks)
- Prefer operation with most successors
Local Scheduling

Forward and backward can produce different results

Latency to the cbr
Subscript to identify
Block from SPEC benchmark "go"

<table>
<thead>
<tr>
<th>Operation</th>
<th>load</th>
<th>loadI</th>
<th>add</th>
<th>addI</th>
<th>store</th>
<th>cmp</th>
</tr>
</thead>
<tbody>
<tr>
<td>Latency</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>

Local Scheduling

Using "latency to root" as the priority function
Scheduling Larger Regions

One step beyond a block is an Extended Basic Block (EBB)

- EBB is a maximal set of blocks s.t.
  - Set has a single entry, $B_i$
  - Each block $B_j$ other than $B_i$ has exactly one predecessor

- Example CFG has three EBBs
  - Big EBB has two paths
    - $(B_1,B_2,B_4)$ & $(B_1,B_3)$

- Many optimizations operate on EBBs
  (including scheduling)
Scheduling Larger Regions

Superlocal Scheduling

- Schedule entire paths through EBBs
- Example has four EBB paths