Instruction Scheduling

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)

**Assumed latencies (conservative)**

<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>loadl</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
  - 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
Example

\[ w \leftarrow w \ast 2 \ast x \ast y \ast z \]

**Simple schedule**                  **Schedule loads early**

<table>
<thead>
<tr>
<th></th>
<th>1 loadAl r0.@w ⇒ r1</th>
<th>1 loadAl r0.@w ⇒ r1</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>add r1,r1 ⇒ r1</td>
<td>2 loadAl r0.@x ⇒ r2</td>
</tr>
<tr>
<td>5</td>
<td>loadAl r0.@x ⇒ r2</td>
<td>3 loadAl r0.@y ⇒ r3</td>
</tr>
<tr>
<td>8</td>
<td>mult r1,r2 ⇒ r1</td>
<td>4 add r1,r1 ⇒ r1</td>
</tr>
<tr>
<td>9</td>
<td>loadAl r0.@y ⇒ r2</td>
<td>5 mult r1,r2 ⇒ r1</td>
</tr>
<tr>
<td>12</td>
<td>mult r1,r2 ⇒ r1</td>
<td>6 loadAl r0.@z ⇒ r2</td>
</tr>
<tr>
<td>13</td>
<td>loadAl r0.@z ⇒ r2</td>
<td>7 mult r1,r3 ⇒ r1</td>
</tr>
<tr>
<td>16</td>
<td>mult r1,r2 ⇒ r1</td>
<td>9 mult r1,r2 ⇒ r1</td>
</tr>
<tr>
<td>18</td>
<td>storeAl r1 ⇒ r0.@w</td>
<td>11 storeAl r1 ⇒ r0.@w</td>
</tr>
</tbody>
</table>
|21 | r1 is free          | 14 r1 is free

2 registers, 20 cycles                  3 registers, 13 cycles

Reordering operations for speed is called instruction scheduling

---

**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**

*Machine description*

- **Slow code**
- **Scheduler**
- **Fast code**

**The task**

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

---

CS430
Instruction Scheduling (The Abstract View)

To capture properties of the code, build a precedence graph \( G \)
- Nodes \( n \in G \) are operations with \( \text{type}(n) \) and \( \text{delay}(n) \)
- An edge \( e = (n_1, n_2) \in G \) if and only if \( n_2 \) uses the result of \( n_1 \)

The Code

\[
\begin{align*}
\text{a: } & \text{loadAI } r0, \text{@w} \Rightarrow r1 \\
\text{b: } & \text{add } r1, r1 \Rightarrow r1 \\
\text{c: } & \text{loadAI } r0, \text{@x} \Rightarrow r2 \\
\text{d: } & \text{mult } r1, r2 \Rightarrow r1 \\
\text{e: } & \text{loadAI } r0, \text{@y} \Rightarrow r2 \\
\text{f: } & \text{mult } r1, r2 \Rightarrow r1 \\
\text{g: } & \text{loadAI } r0, \text{@z} \Rightarrow r2 \\
\text{h: } & \text{mult } r1, r2 \Rightarrow r1 \\
\text{i: } & \text{storeAI } r1 \Rightarrow r0, \text{@w}
\end{align*}
\]

The Precedence Graph

\[
\begin{align*}
a & \rightarrow b \\
b & \rightarrow c, d, e \\
c & \rightarrow a, d, e \\
d & \rightarrow c, f \\
e & \rightarrow a, c, d, f \\
f & \rightarrow d, e, g \\
g & \rightarrow f, h \\
h & \rightarrow g, i \\
i & \rightarrow h 
\end{align*}
\]

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

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, one cycle at a time
   a. Use a queue of operations that are ready
   b. At each cycle
      I. Choose a ready operation and schedule it
      II. Update the ready queue

Local list scheduling
• The dominant algorithm for twenty years
• A greedy, heuristic, local technique
Local List Scheduling

\[
\text{Cycle} \leftarrow 1 \\
\text{Ready} \leftarrow \text{leaves of } P \\
\text{Active} \leftarrow \emptyset \\
\text{while } (\text{Ready} \cup \text{Active} \neq \emptyset) \\
\quad \text{if } (\text{Ready} \neq \emptyset) \text{ then} \\
\quad \quad \text{remove an op from Ready} \\
\quad \quad \text{S(op)} \leftarrow \text{Cycle} \\
\quad \quad \text{Active} \leftarrow \text{Active} \cup \text{op} \\
\quad \text{Cycle} \leftarrow \text{Cycle} + 1 \\
\quad \text{for each op \in Active} \\
\quad \qquad \text{if } (\text{S(op)} + \text{delay(op)} \leq \text{Cycle}) \text{ then} \\
\quad \qquad \quad \text{remove op from Active} \\
\quad \qquad \text{for each successor s of op in } P \\
\quad \qquad \quad \text{if } (s \text{ is ready}) \text{ then} \\
\quad \qquad \quad \quad \text{Ready} \leftarrow \text{Ready} \cup s
\]

Removal in priority order

op has completed execution

If successor’s operands are ready, put it on Ready

Scheduling Example

1. Build the precedence graph

\[
\begin{align*}
\text{a: loadAl} & \quad r_0, @w \Rightarrow r_1 \\
\text{b: add} & \quad r_1, r_1 \Rightarrow r_1 \\
\text{c: loadAl} & \quad r_0, @x \Rightarrow r_2 \\
\text{d: mult} & \quad r_1, r_2 \Rightarrow r_1 \\
\text{e: loadAl} & \quad r_0, @y \Rightarrow r_2 \\
\text{f: mult} & \quad r_1, r_2 \Rightarrow r_1 \\
\text{g: loadAl} & \quad r_0, @z \Rightarrow r_2 \\
\text{h: mult} & \quad r_1, r_2 \Rightarrow r_1 \\
\text{i: storeAl} & \quad r_1 \Rightarrow r_0, @w
\end{align*}
\]

The Code

The Precedence Graph
Scheduling Example

1. Build the precedence graph
2. Determine priorities: longest latency-weighted path

a: loadAI r0,@w ⇒ r1
b: add r1,r1 ⇒ r1
c: loadAI r0,@x ⇒ r2
d: mult r1,r2 ⇒ r1
e: loadAI r0,@y ⇒ r2
f: mult r1,r2 ⇒ r1
g: loadAI r0,@z ⇒ r2
h: mult r1,r2 ⇒ r1
i: storeAI r1 ⇒ r0,@w

The Code

The Precedence Graph

New register name used
Detailed Scheduling Algorithm I

Idea: Keep a collection of worklists \( W[c] \), one per cycle

\( \rightarrow \) We need \( MaxC = \text{max delay} + 1 \) such worklists

Code:

```plaintext
for each \( n \in N \) do begin
  \( \text{count}[n] := 0; \text{earliest}[n] = 0 \)
end

for each \( (n1,n2) \in E \) do begin
  \( \text{count}[n2] := \text{count}[n2] + 1; \)
  \( \text{successors}[n1] := \text{successors}[n1] \cup \{n2\}; \)
end

for i := 0 to \( MaxC - 1 \) do
  \( W[i] := \emptyset; \)
  \( \text{Wcount} := 0; \)
for each \( n \in N \)
  if \( \text{count}[n] = 0 \) then begin
    \( W[0] := W[0] \cup \{n\}; \text{Wcount} := \text{Wcount} + 1; \)
  end

\( c := 0; \) // \( c \) is the cycle number
\( cW := 0; \) // \( cW \) is the number of the worklist for cycle \( c \)
\( \text{instr}[c] := \emptyset; \)

Idea: Keep a collection of worklists \( W[c] \), one per cycle

\( \rightarrow \) We need \( MaxC = \text{max delay} + 1 \) such worklists

Code:

```plaintext
while \( \text{Wcount} > 0 \) do begin
  while \( W[cW] = \emptyset \) do begin
    \( c := c + 1; \text{instr}[c] := \emptyset; cW := \text{mod}(cW+1,MaxC); \)
  end
  \( \text{nextc} := \text{mod}(c+1,MaxC); \)
  while \( W[cW] \neq \emptyset \) do begin
    \( \text{select and remove an arbitrary instruction } x \text{ from } W[cW]; \)
    if \( \exists \) free issue units of type(\( x \)) on cycle \( c \) then begin
      \( \text{instr}[c] := \text{instr}[c] \cup \{x\}; \text{Wcount} := \text{Wcount} - 1; \)
      for each \( y \in \text{successors}[x] \) do begin
        \( \text{count}[y] := \text{count}[y] - 1; \)
        \( \text{earliest}[y] := \max(\text{earliest}[y], c+\text{delay}(x)); \)
        if \( \text{count}[y] = 0 \) then begin
          \( \text{loc} := \text{mod}(\text{earliest}[y],MaxC); \)
          \( W[\text{loc}] := W[\text{loc}] \cup \{y\}; \text{Wcount} := \text{Wcount} + 1; \)
        end
      end
    end
    \( \text{else } W[\text{nextc}] := W[\text{nextc}] \cup \{x\}; \)
  end
end
```

Priority —— select and remove an arbitrary instruction \( x \) from \( W[cW] \);

If \( \exists \) free issue units of type(\( x \)) on cycle \( c \) then begin

\( \text{instr}[c] := \text{instr}[c] \cup \{x\}; \text{Wcount} := \text{Wcount} - 1; \)

for each \( y \in \text{successors}[x] \) do begin

\( \text{count}[y] := \text{count}[y] - 1; \)

\( \text{earliest}[y] := \max(\text{earliest}[y], c+\text{delay}(x)); \)

if \( \text{count}[y] = 0 \) then begin

\( \text{loc} := \text{mod}(\text{earliest}[y],\text{MaxC}); \)

\( W[\text{loc}] := W[\text{loc}] \cup \{y\}; \text{Wcount} := \text{Wcount} + 1; \)

end

\end
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