Local Instruction Scheduling
— A Primer for Lab 3 —

Comp 412

Copyright 2018, Keith D. Cooper & Linda Torczon, all rights reserved.
Students enrolled in Comp 412 at Rice University have explicit permission to make copies of these materials for their personal use.
Faculty from other educational institutions may use these materials for nonprofit educational purposes, provided this copyright notice is preserved.

Chapter 12 in EaC2e
## Lab 3 Schedule

<table>
<thead>
<tr>
<th>Sunday</th>
<th>Monday</th>
<th>Tuesday</th>
<th>Wednesday</th>
<th>Thursday</th>
<th>Friday</th>
<th>Saturday</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oct 21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
</tr>
</tbody>
</table>

- **Docs Available Lab Lecture**
- **Start work**
- **Complete & Correct Dependence Graph**
- **Correct Schedule**
- **Improve Schedules**

**Lab 3 Due Date**

**No Class (Walk)**

**Thanksgiving Break**

**Late day**

**Not a late day**

**Not a late day**

**Not a late day**

**You are here**

**The goal is here**
Order of Operations Matters

• 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)

Instruction Scheduling reorders operations
• Minimize wasted cycles (cover latencies with useful work)
• Maximize functional unit utilization

Opportunities
• Non-blocking operations
  – Can issue new ops while waiting for one that is executing
• Operations that are independent of each other
  – Can issue independent operations in parallel with each other
To schedule operations, the compiler needs accurate data on latencies. This data is surprisingly hard to measure accurately:

- **Value-dependent behavior**
  - Multiply depends on bit patterns

- **Context-dependent behavior**

- **Compiler behavior**
  - Have seen gcc underallocate & introduce excess spills & their costs
  - 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>
Order of Operations Matters

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

Instruction Scheduling reorders operations

• Minimize wasted cycles (cover latencies with useful work)
• Maximize functional unit utilization

Opportunities

• Non-blocking operations
  – Can issue new ops while waiting for one that is executing
• Operations that are independent of each other
  – Can issue independent operations in parallel with each other

In Lab 3, you will build a local instruction scheduler for straight-line code in ILOC.
Example from Chapter 1

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

<table>
<thead>
<tr>
<th>start</th>
<th>end</th>
<th>Naïve Schedule</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>loadAI r_{ARP}, @a \Rightarrow r1</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>add r1, r1 \Rightarrow r1</td>
</tr>
<tr>
<td>5</td>
<td>7</td>
<td>loadAI r_{ARP}, @b \Rightarrow r2</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>mult r1, r2 \Rightarrow r1</td>
</tr>
<tr>
<td>10</td>
<td>12</td>
<td>loadAI r_{ARP}, @c \Rightarrow r2</td>
</tr>
<tr>
<td>13</td>
<td>14</td>
<td>mult r1, r2 \Rightarrow r1</td>
</tr>
<tr>
<td>15</td>
<td>17</td>
<td>loadAI r_{ARP}, @d \Rightarrow r2</td>
</tr>
<tr>
<td>18</td>
<td>19</td>
<td>mult r1, r2 \Rightarrow r1</td>
</tr>
<tr>
<td>20</td>
<td>22</td>
<td>storeAI r1 \Rightarrow r_{ARP}, @a</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Op</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>loads &amp; stores</td>
<td>3 cycles</td>
</tr>
<tr>
<td>multiplies</td>
<td>2 cycles</td>
</tr>
<tr>
<td>all others</td>
<td>1 cycle</td>
</tr>
</tbody>
</table>

- Schedule for a single functional unit
- Naïve schedule takes ops in treewalk order

\[ \leftarrow \text{These are the lecture 1 latencies, not the lab 3 latencies} \]
Example from Chapter 1

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

<table>
<thead>
<tr>
<th>start</th>
<th>end</th>
<th>Naïve Schedule</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>load AI ( r_{\text{ARP}, @a} \Rightarrow r1 )</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>add ( r1, r1 \Rightarrow r1 )</td>
</tr>
<tr>
<td>5</td>
<td>7</td>
<td>load AI ( r_{\text{ARP}, @b} \Rightarrow r2 )</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>mult ( r1, r2 \Rightarrow r1 )</td>
</tr>
<tr>
<td>10</td>
<td>12</td>
<td>load AI ( r_{\text{ARP}, @c} \Rightarrow r2 )</td>
</tr>
<tr>
<td>13</td>
<td>14</td>
<td>mult ( r1, r2 \Rightarrow r1 )</td>
</tr>
<tr>
<td>15</td>
<td>17</td>
<td>load AI ( r_{\text{ARP}, @d} \Rightarrow r2 )</td>
</tr>
<tr>
<td>18</td>
<td>19</td>
<td>mult ( r1, r2 \Rightarrow r1 )</td>
</tr>
<tr>
<td>20</td>
<td>22</td>
<td>store AI ( r1 \Rightarrow r_{\text{ARP}, @a} )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>start</th>
<th>end</th>
<th>Schedule Loads Early</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>load AI ( r_{\text{ARP}, @a} \Rightarrow r1 )</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>load AI ( r_{\text{ARP}, @b} \Rightarrow r2 )</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>load AI ( r_{\text{ARP}, @c} \Rightarrow r3 )</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>add ( r1, r1 \Rightarrow r1 )</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>mult ( r1, r2 \Rightarrow r1 )</td>
</tr>
<tr>
<td>6</td>
<td>8</td>
<td>load AI ( r_{\text{ARP}, @d} \Rightarrow r2 )</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>mult ( r1, r3 \Rightarrow r1 )</td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td>mult ( r1, r2 \Rightarrow r1 )</td>
</tr>
<tr>
<td>11</td>
<td>13</td>
<td>store AI ( r1 \Rightarrow r_{\text{ARP}, @a} )</td>
</tr>
</tbody>
</table>

- Schedule for a single functional unit
- Naïve schedule takes ops in treewalk order
- Reordered code overlaps latencies in the load operations
- Shows potential of non-blocking operations

<table>
<thead>
<tr>
<th>Op</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>loads &amp; stores</td>
<td>3 cycles</td>
</tr>
<tr>
<td>multiplies</td>
<td>2 cycles</td>
</tr>
<tr>
<td>all others</td>
<td>1 cycle</td>
</tr>
</tbody>
</table>

1 more register
1 cycle stall
Reuse of register names severely constrains our ability to reorder the instructions.

The solution: rename into live ranges as the first step in your scheduler.

For Lab 3, you have “enough” registers in the simulator.

(You have the code from lab 2. Remember code check 1?)
Instruction Scheduling  (Black Box Model)

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*

```plaintext
slow code \downarrow \quad \text{Scheduler} \quad \downarrow fast code
```

The Task

- Produce correct code
- Minimize wasted cycles
- Avoid spilling registers
- Operate efficiently
1. Rename registers into live ranges to eliminate artificial constraints
   • Reuse code from lab 1
   • Every definition creates a new name (*avoids constraint from example*)

The Original Code

a: loadAl \( r_{ARP, @a} \)  \( \Rightarrow \) r1
b: add \( r1, r1 \)  \( \Rightarrow \) r1
c: loadAl \( r_{ARP, @b} \)  \( \Rightarrow \) r2
d: mult \( r1, r2 \)  \( \Rightarrow \) r1
e: loadAl \( r_{ARP, @c} \)  \( \Rightarrow \) r2
f: mult \( r1, r2 \)  \( \Rightarrow \) r1
g: loadAl \( r_{ARP, @d} \)  \( \Rightarrow \) r2
h: mult \( r1, r2 \)  \( \Rightarrow \) r1
i: storeAl \( r1 \)  \( \Rightarrow \) r0,@a
### Instruction Scheduling (Operational View)

#### 1. Rename registers into live ranges to eliminate artificial constraints

- Every definition creates a unique name
- Reuse of source names might prevent some otherwise legal orders

<table>
<thead>
<tr>
<th>Original Code</th>
<th>After Renaming</th>
</tr>
</thead>
<tbody>
<tr>
<td>a: <code>loadAl r_{ARP}, @a \Rightarrow r1</code></td>
<td>a: <code>loadAl r_{ARP}, @a \Rightarrow r1</code></td>
</tr>
<tr>
<td>b: <code>add r1, r1 \Rightarrow r1</code></td>
<td>b: <code>add r1, r1 \Rightarrow r2</code></td>
</tr>
<tr>
<td>c: <code>loadAl r_{ARP}, @b \Rightarrow r2</code></td>
<td>c: <code>loadAl r_{ARP}, @b \Rightarrow r3</code></td>
</tr>
<tr>
<td>d: <code>mult r1, r2 \Rightarrow r1</code></td>
<td>d: <code>mult r2, r3 \Rightarrow r4</code></td>
</tr>
<tr>
<td>e: <code>loadAl r_{ARP}, @c \Rightarrow r2</code></td>
<td>e: <code>loadAl r_{ARP}, @c \Rightarrow r5</code></td>
</tr>
<tr>
<td>f: <code>mult r1, r2 \Rightarrow r1</code></td>
<td>f: <code>mult r4, r5 \Rightarrow r6</code></td>
</tr>
<tr>
<td>g: <code>loadAl r_{ARP}, @d \Rightarrow r2</code></td>
<td>g: <code>loadAl r_{ARP}, @d \Rightarrow r7</code></td>
</tr>
<tr>
<td>h: <code>mult r1, r2 \Rightarrow r1</code></td>
<td>h: <code>mult r6, r7 \Rightarrow r8</code></td>
</tr>
<tr>
<td>i: <code>storeAl r1 \Rightarrow r0, @a</code></td>
<td>i: <code>storeAl r8 \Rightarrow r0, @a</code></td>
</tr>
</tbody>
</table>

The Original Code

The text states that renaming assumes "enough" registers. You have them.
Instruction Scheduling  (Operational View)

2. Build a dependence graph to capture critical relationships in the code
   - Nodes $n \in D$ are operations with $\text{type}(n)$ and $\text{delay}(n)$
   - Has an edge $e = (n_1, n_2) \in D$ if $n_1$ uses the result of $n_2$
     - Will have some additional edges, as well

\[\begin{align*}
a & : \text{loadAl} \quad r_{\text{ARP}, @a} \Rightarrow r1 \\
b & : \text{add} \quad r1, r1 \Rightarrow r2 \\
c & : \text{loadAl} \quad r_{\text{ARP}, @b} \Rightarrow r3 \\
d & : \text{mult} \quad r2, r3 \Rightarrow r4 \\
e & : \text{loadAl} \quad r_{\text{ARP}, @c} \Rightarrow r5 \\
f & : \text{mult} \quad r4, r5 \Rightarrow r6 \\
g & : \text{loadAl} \quad r_{\text{ARP}, @d} \Rightarrow r7 \\
h & : \text{mult} \quad r6, r7 \Rightarrow r8 \\
i & : \text{storeAl} \quad r8 \Rightarrow r0, @a
\end{align*}\]

After Renaming

The Dependence Graph

Some authors prefer the term “precedence graph” to “dependence graph.” They are, in the context of scheduling, interchangeable.
2. Build a **dependence graph** to capture critical relationships in the code

- The final schedule must honor each dependence in the input code
  - *Values computed before they are used & used before they are rewritten*
- Renaming lets the scheduler ignore some (anti) dependences

```
| a: loadAl, r_{ARP}, @a => r1 |
| b: add, r1, r1 => r2 |
| c: loadAl, r_{ARP}, @b => r3 |
| d: mult, r2, r3 => r4 |
| e: loadAl, r_{ARP}, @c => r5 |
| f: mult, r4, r5 => r6 |
| g: loadAl, r_{ARP}, @d => r7 |
| h: mult, r6, r7 => r8 |
| i: storeAl, r8 => r0, @a |
```

**After Renaming**

**The Dependence Graph**
The Scheduler Needs Some Additional Edges

To ensure the correct flow of values, the scheduler needs edges that specify the relative ordering of loads, stores, and outputs

<table>
<thead>
<tr>
<th>First op</th>
<th>Second op</th>
<th>Same address</th>
<th>Distinct addresses</th>
<th>Unknown address(es)</th>
<th>Acronym</th>
</tr>
</thead>
<tbody>
<tr>
<td>load</td>
<td>load</td>
<td>— No conflict <em>(after renaming)</em> —</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>load or output</td>
<td>store</td>
<td>conflict</td>
<td>no conflict</td>
<td>conflict</td>
<td>WAR</td>
</tr>
<tr>
<td>store</td>
<td>store</td>
<td>conflict</td>
<td>no conflict</td>
<td>conflict</td>
<td>WAW</td>
</tr>
<tr>
<td>store</td>
<td>load or output</td>
<td>conflict</td>
<td>no conflict</td>
<td>conflict</td>
<td>RAW</td>
</tr>
<tr>
<td>output</td>
<td>output</td>
<td>need an edge to serialize the outputs</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

"conflict" implies 1st op must complete before 2nd op issues
⇒ an edge in dependence graph
2. Build a dependence graph to capture critical relationships in the code

Create an empty map, \( M \)
walk the block, top to bottom
at each operation \( o \) that defines \( VR_i \):
create a node for \( o \)
set \( M(VR_i) \) to \( o \)
for each \( VR_j \) used in \( o \), add an edge from \( o \) to the node in \( M(VR_j) \)
if \( o \) is a load, store, or output
operation, add edges to ensure serialization of memory ops

\( O(n + m^2) \)


Building the Graph

Explanatory Notes

1. ‘\( o \)’ refers to both the operation in the original block and the node that represents it in the graph.
The meaning should be clear by context.

2. Additional edges to ensure correctness:
   - load & output need an edge to the most recent store (full latency)
   - output needs an edge to the most recent output (serialization)
   - store needs an edge to the most recent store, as well as each previous load & output (serialization)

If your lab tries to simplify the graph, it may need more edges for store & output nodes

You can also formulate the dependence-graph builder as an extension of the algorithm for finding live ranges in a block (lecture 3, slide 15).
3. **Compute a priority function on the nodes of the graph**

   - The priority function should reflect the importance of scheduling that operation early in the block
     - *Latency-weighted distance to a root is a classic priority function*

   a: loadAI $r_{ARP}, @a \Rightarrow r1$
   b: add $r1, r1 \Rightarrow r2$
   c: loadAI $r_{ARP}, @b \Rightarrow r3$
   d: mult $r2, r3 \Rightarrow r4$
   e: loadAI $r_{ARP}, @c \Rightarrow r5$
   f: mult $r4, r5 \Rightarrow r6$
   g: loadAI $r_{ARP}, @d \Rightarrow r7$
   h: mult $r6, r7 \Rightarrow r8$
   i: storeAI $r8 \Rightarrow r0, @a$

**After Renaming**

**The Dependence Graph**
3. Compute a priority function on the nodes of the graph
   
   The priority function should reflect the importance of scheduling that operation early in the block
   
   - Latency-weighted distance to a root is a classic priority function

<table>
<thead>
<tr>
<th>Op</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>loads &amp; stores</td>
<td>3 cycles</td>
</tr>
<tr>
<td>multiplies</td>
<td>2 cycles</td>
</tr>
<tr>
<td>all others</td>
<td>1 cycle</td>
</tr>
</tbody>
</table>
4. Schedule the operations to reflect dependences and priorities

\[
\begin{align*}
\text{Cycle} & \leftarrow 1 \\
\text{Ready} & \leftarrow \text{leaves of } D \\
\text{Active} & \leftarrow \emptyset \\
\end{align*}
\]

while (Ready \cup Active \neq \emptyset)

\[
\begin{align*}
\text{for each functional unit, } f, \text{ do} \\
& \quad \text{if } \exists 1 \text{ or more } op \text{ in Ready for } f \text{ then} \\
& \quad \quad \text{remove highest prio. } op \text{ for } f \text{ from Ready} \\
& \quad \quad S(op) \leftarrow \text{Cycle} \\
& \quad \quad \text{Active} \leftarrow \text{Active } \cup \text{op} \\
\end{align*}
\]

\[
\text{Cycle} \leftarrow \text{Cycle} + 1
\]

\[
\begin{align*}
\text{for each } op \text{ in Active} \\
& \quad \text{if } (S(op) + \text{delay}(op) \leq \text{Cycle}) \text{ then} \\
& \quad \quad \text{remove } op \text{ from Active} \\
& \quad \quad \text{add the } \text{ready} \text{ successors of } op \text{ to Ready} \\
& \quad \text{if } ((op \text{ is load or store}) \& S(op) < \text{cycle}) \\
& \quad \quad \text{add the } \text{ready} \text{ successors of } op \text{ to Ready}
\end{align*}
\]

Think of Ready as a priority queue.

Remove in priority order (\textit{break ties})

\textit{op} has completed execution

for (s in successors(op))

\[
\begin{align*}
\text{if } (s \text{ is ready}) \\
& \quad \text{Ready} \leftarrow \text{Ready } \cup \text{op}
\end{align*}
\]

Does not quite fit inline
Instruction Scheduling  (Operational View)

4. Schedule the operations to reflect dependences and priorities

Test for “s is ready” is a little complex

- Need to check whether each constraint, represented by an edge to a predecessor, has been satisfied
- Could go to successor s, then look back up each edge and check to see if that predecessor op has completed (for a dependence edge) or the op has issued (serialization edge)
- Could, instead, mark each edge when it is satisfied & then iterate over the edges entering successor s are all marked as satisfied
- Could “ref count” satisfied edges

```
Cycle ← Cycle + 1

for each op in Active
    if (S(op) + delay(op) ≤ Cycle) then
        remove op from Active
    for (s in successors(op))
        if (s is ready)
            Ready ← Ready ∪ op

if ((op is load or store) & S(op) < cycle)
    for (s in successors(op))
        if (s is ready)
            Ready ← Ready ∪ op
```
4. Schedule the operations to reflect dependences and priorities
   • Use a greedy heuristic technique, such as *list scheduling*

Start with the operations that are ready to schedule (the leaves)
   — remove completed ops from Active and visit ops that depend on them. If they are ready, move them onto the Ready queue
   — take the highest priority op in Ready & schedule it into next slot
   — place the newly schedule op in Active

The Schedule

<table>
<thead>
<tr>
<th>Ready</th>
<th>&lt;a,13&gt;, &lt;c,12&gt;, &lt;e,10&gt;, &lt;g,8&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Active</td>
<td></td>
</tr>
</tbody>
</table>

Dependence Graph

Schedule

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
</table>
Instruction Scheduling

4. Schedule the operations to reflect dependences and priorities
   • Use a greedy heuristic technique, such as list scheduling

Start with the operations that are ready to schedule (the leaves)
   — remove completed ops from Active and visit ops that depend on them. If they are ready, move them onto the Ready queue
   — take the highest priority op in Ready & schedule it into next slot
   — place the newly schedule op in Active

The Schedule

<table>
<thead>
<tr>
<th>Schedule</th>
<th>Ready</th>
<th>Active</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>&lt;c,12&gt;, &lt;e,10&gt;, &lt;g,8&gt;</td>
<td>&lt;a,#4&gt;</td>
</tr>
<tr>
<td>1</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
4. Schedule the operations to reflect dependences and priorities

- Use a greedy heuristic technique, such as *list scheduling*

**Start with the operations that are ready to schedule** (*the leaves*)

- remove completed ops from Active and visit ops that depend on them. If they are ready, move them onto the Ready queue

- take the highest priority op in Ready & schedule it into next slot

- place the newly schedule op in Active
4. Schedule the operations to reflect dependences and priorities

- Use a greedy heuristic technique, such as *list scheduling*

<table>
<thead>
<tr>
<th>Schedule</th>
<th>1</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>c</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>e</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>5</td>
<td></td>
</tr>
<tr>
<td></td>
<td>6</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7</td>
<td></td>
</tr>
<tr>
<td></td>
<td>8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>9</td>
<td></td>
</tr>
<tr>
<td></td>
<td>10</td>
<td></td>
</tr>
<tr>
<td></td>
<td>11</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Ready</th>
<th>&lt;g,8&gt;, &lt;b,10&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Active</td>
<td>&lt;a,#4&gt;, &lt;c,#5&gt;, &lt;e,#6&gt;</td>
</tr>
</tbody>
</table>

Start with the operations that are ready to schedule (the leaves):
- remove completed ops from Active and visit ops that depend on them. If they are ready, move them onto the Ready queue
- take the highest priority op in Ready & schedule it into next slot
- place the newly schedule op in Active

**The Schedule**
4. Schedule the operations to reflect dependences and priorities
   • Use a greedy heuristic technique, such as list scheduling

Start with the operations that are ready to schedule (the leaves)
   — remove completed ops from Active and visit ops that depend on them. If they are ready, move them onto the Ready queue
   — take the highest priority op in Ready & schedule it into next slot
   — place the newly schedule op in Active

**The Schedule**

<table>
<thead>
<tr>
<th>Schedule</th>
<th>1</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>c</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>e</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>5</td>
<td></td>
</tr>
<tr>
<td></td>
<td>6</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7</td>
<td></td>
</tr>
<tr>
<td></td>
<td>8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>9</td>
<td></td>
</tr>
<tr>
<td></td>
<td>10</td>
<td></td>
</tr>
<tr>
<td></td>
<td>11</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Ready</th>
<th>&lt;g,8&gt;, &lt;b,10&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Active</td>
<td>&lt;a,#4&gt;, &lt;c,#5&gt;, &lt;e,#6&gt;</td>
</tr>
</tbody>
</table>
4. Schedule the operations to reflect dependences and priorities

- Use a greedy heuristic technique, such as list scheduling

Start with the operations that are ready to schedule (the leaves)
- remove completed ops from Active and visit ops that depend on them. If they are ready, move them onto the Ready queue
- take the highest priority op in Ready & schedule it into next slot
- place the newly schedule op in Active

The Schedule
Instruction Scheduling

4. Schedule the operations to reflect dependences and priorities
   • Use a greedy heuristic technique, such as list scheduling

Start with the operations that are ready to schedule (the leaves)
   — remove completed ops from Active and visit ops that depend on them. If they are ready, move them onto the Ready queue
   — take the highest priority op in Ready & schedule it into next slot
   — place the newly schedule op in Active

The Schedule

<table>
<thead>
<tr>
<th>Schedule</th>
<th>Ready</th>
<th>Active</th>
</tr>
</thead>
<tbody>
<tr>
<td>1  a</td>
<td>&lt;g,8&gt;,</td>
<td>&lt;c,#5&gt;, &lt;e,#6&gt;, &lt;b,#5&gt;</td>
</tr>
<tr>
<td>2  c</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3  e</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4  b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5  d</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6  e</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7  f</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8  g</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9  h</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10 e</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11 i</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Instruction Scheduling

4. Schedule the operations to reflect dependences and priorities
   • Use a greedy heuristic technique, such as list scheduling

Start with the operations that are ready to schedule (the leaves)
   — remove completed ops from Active and visit ops that depend on them. If they are ready, move them onto the Ready queue
   — take the highest priority op in Ready & schedule it into next slot
   — place the newly scheduled op in Active

<table>
<thead>
<tr>
<th>Schedule</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>11</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Ready</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;g,8&gt;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Active</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;e,#6&gt;, &lt;d, #7&gt;</td>
</tr>
</tbody>
</table>
Instruction Scheduling

4. Schedule the operations to reflect dependences and priorities
   • Use a greedy heuristic technique, such as list scheduling

Start with the operations that are ready to schedule (the leaves)
   — remove completed ops from Active and visit ops that depend on them. If they are ready, move them onto the Ready queue
   — take the highest priority op in Ready & schedule it into next slot
   — place the newly schedule op in Active
Instruction Scheduling

4. Schedule the operations to reflect dependences and priorities
   • Use a greedy heuristic technique, such as list scheduling

```
Start with the operations that are ready to schedule (the leaves)
   — remove completed ops from Active and visit ops that depend on them. If they are ready, move them onto the Ready queue
   — take the highest priority op in Ready & schedule it into next slot
   — place the newly schedule op in Active

<table>
<thead>
<tr>
<th>Schedule</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>11</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Ready</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;g,#9&gt;, &lt;d,#7&gt;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Active</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

Dependence Graph

The Schedule

COMP 412, Fall 2018

28
4. Schedule the operations to reflect dependences and priorities

- Use a greedy heuristic technique, such as *list scheduling*

**Dependence Graph**

**Schedule**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
</tr>
<tr>
<td>2</td>
<td>c</td>
</tr>
<tr>
<td>3</td>
<td>e</td>
</tr>
<tr>
<td>4</td>
<td>b</td>
</tr>
<tr>
<td>5</td>
<td>d</td>
</tr>
<tr>
<td>6</td>
<td>g</td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
</tr>
</tbody>
</table>

**Ready**

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;f,7&gt;</td>
</tr>
</tbody>
</table>

**Active**

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;g,#9&gt;, &lt;d,#7&gt;</td>
</tr>
</tbody>
</table>

Start with the operations that are ready to schedule (*the leaves*)

- remove completed ops from Active and visit ops that depend on them. If they are ready, move them onto the Ready queue

- take the highest priority op in Ready & schedule it into next slot

- place the newly schedule op in Active

**The Schedule**
4. Schedule the operations to reflect dependences and priorities

- Use a greedy heuristic technique, such as list scheduling

Start with the operations that are ready to schedule (the leaves)
- remove completed ops from Active and visit ops that depend on them. If they are ready, move them onto the Ready queue
- take the highest priority op in Ready & schedule it into next slot
- place the newly schedule op in Active
Instruction Scheduling

4. Schedule the operations to reflect dependences and priorities
   • Use a greedy heuristic technique, such as list scheduling

Dependence Graph

Schedule

<table>
<thead>
<tr>
<th>Schedule</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>11</td>
</tr>
</tbody>
</table>

Ready | nothing ready |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Active</td>
<td>&lt;g, #9&gt;, &lt;f, #9&gt;</td>
</tr>
</tbody>
</table>

Start with the operations that are ready to schedule (the leaves)
   - remove completed ops from Active and visit ops that depend on them. If they are ready, move them onto the Ready queue
   - take the highest priority op in Ready & schedule it into next slot
   - place the newly schedule op in Active

The Schedule
4. Schedule the operations to reflect dependences and priorities

- Use a greedy heuristic technique, such as *list scheduling*

**Dependence Graph**

**The Schedule**

<table>
<thead>
<tr>
<th>Schedule</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>11</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Ready</th>
<th>nothing ready</th>
</tr>
</thead>
<tbody>
<tr>
<td>Active</td>
<td>&lt;g, #9&gt;, &lt;f,#9&gt;</td>
</tr>
</tbody>
</table>

Start with the operations that are ready to schedule (*the leaves*)

- remove completed ops from Active and visit ops that depend on them. If they are ready, move them onto the Ready queue
- take the highest priority op in Ready & schedule it into next slot
- place the newly scheduled op in Active
Instruction Scheduling

4. Schedule the operations to reflect dependences and priorities
   • Use a greedy heuristic technique, such as *list scheduling*

### Dependence Graph

#### Schedule

<table>
<thead>
<tr>
<th>Schedule</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>11</td>
</tr>
</tbody>
</table>

#### Ready

<table>
<thead>
<tr>
<th>Ready</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;h,5&gt;</td>
</tr>
</tbody>
</table>

#### Active

<table>
<thead>
<tr>
<th>Active</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;g,#9&gt;, &lt;f,#9&gt;</td>
</tr>
</tbody>
</table>

Start with the operations that are ready to schedule (*the leaves*)

- remove completed ops from Active and visit ops that depend on them. If they are ready, move them onto the Ready queue
- take the highest priority op in Ready & schedule it into next slot
- place the newly schedule op in Active

### The Schedule
Instruction Scheduling

4. Schedule the operations to reflect dependences and priorities
   • Use a greedy heuristic technique, such as list scheduling

Start with the operations that are ready to schedule (the leaves)
   — remove completed ops from Active and visit ops that depend on them. If they are ready, move them onto the Ready queue
   — take the highest priority op in Ready & schedule it into next slot
   — place the newly schedule op in Active

<table>
<thead>
<tr>
<th>Schedule</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>11</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Ready</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Active</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;h,#11&gt;</td>
</tr>
</tbody>
</table>

Dependence Graph

The Schedule
4. Schedule the operations to reflect dependences and priorities

- Use a greedy heuristic technique, such as *list scheduling*

Start with the operations that are ready to schedule (the leaves):
- remove completed ops from Active and visit ops that depend on them. If they are ready, move them onto the Ready queue
- take the highest priority op in Ready & schedule it into next slot
- place the newly schedule op in Active

---

**Schedule**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
</tr>
<tr>
<td>2</td>
<td>c</td>
</tr>
<tr>
<td>3</td>
<td>e</td>
</tr>
<tr>
<td>4</td>
<td>b</td>
</tr>
<tr>
<td>5</td>
<td>d</td>
</tr>
<tr>
<td>6</td>
<td>g</td>
</tr>
<tr>
<td>7</td>
<td>f</td>
</tr>
<tr>
<td>8</td>
<td>nop</td>
</tr>
<tr>
<td>9</td>
<td>h</td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
</tr>
</tbody>
</table>

**Ready**

- nothing ready

**Active**

- <h,#11>
Instruction Scheduling

4. Schedule the operations to reflect dependences and priorities
   • Use a greedy heuristic technique, such as list scheduling

Start with the operations that are ready to schedule (the leaves)
   — remove completed ops from Active and visit ops that depend on them. If they are ready, move them onto the Ready queue
   — take the highest priority op in Ready & schedule it into next slot
   — place the newly schedule op in Active
4. Schedule the operations to reflect dependences and priorities
   • Use a greedy heuristic technique, such as list scheduling

Start with the operations that are ready to schedule (the leaves)
   — remove completed ops from Active and visit ops that depend on them. If they are ready, move them onto the Ready queue
   — take the highest priority op in Ready & schedule it into next slot
   — place the newly schedule op in Active
4. Schedule the operations to reflect dependences and priorities

- Use a greedy heuristic technique, such as *list scheduling*

---

**Dependence Graph**

**Schedule**

<table>
<thead>
<tr>
<th>Number</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
</tr>
<tr>
<td>2</td>
<td>c</td>
</tr>
<tr>
<td>3</td>
<td>e</td>
</tr>
<tr>
<td>4</td>
<td>b</td>
</tr>
<tr>
<td>5</td>
<td>d</td>
</tr>
<tr>
<td>6</td>
<td>g</td>
</tr>
<tr>
<td>7</td>
<td>f</td>
</tr>
<tr>
<td>8</td>
<td>nop</td>
</tr>
<tr>
<td>9</td>
<td>h</td>
</tr>
<tr>
<td>10</td>
<td>nop</td>
</tr>
<tr>
<td>11</td>
<td>i</td>
</tr>
</tbody>
</table>

**Ready**

<table>
<thead>
<tr>
<th>Number</th>
<th>Operation</th>
</tr>
</thead>
</table>

**Active**

---

Start with the operations that are ready to schedule (*the leaves*)

- remove completed ops from Active and visit ops that depend on them. If they are ready, move them onto the Ready queue
- take the highest priority op in Ready & schedule it into next slot
- place the newly scheduled op in Active

---

**The Schedule**

COMP 412, Fall 2018
Instruction Scheduling

4. Schedule the operations to reflect dependences and priorities
   • Use a greedy heuristic technique, such as list scheduling

Start with the operations that are ready to schedule \((\text{the leaves})\)
— remove completed ops from Active and visit ops that depend on them.
If they are ready, move them onto the Ready queue
— take the highest priority op in Ready & schedule it into next slot
— place the newly scheduled op in Active

And, the ready queue and the active queue are empty, at last … so, it halts.
Same schedule as on slide 5.

```
<table>
<thead>
<tr>
<th>Schedule</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>11</td>
</tr>
</tbody>
</table>
```

Ready
Active

Dependence Graph
Cycle $\leftarrow 1$
Ready $\leftarrow$ leaves of $D$
Active $\leftarrow \emptyset$

while (Ready $\cup$ Active $\neq \emptyset$)
    for each functional unit, $f$, do
        if $\exists$ 1 or more $op$ in Ready for $f$ then
            remove highest prio. $op$ for $f$ from Ready
            $S(op) \leftarrow$ Cycle
            Active $\leftarrow$ Active $\cup$ $op$

    Cycle $\leftarrow$ Cycle + 1

    for each $op$ in Active
        if ($S(op) + \text{delay}(op) \leq \text{Cycle}$) then
            remove $op$ from Active
            add the ready successors of $op$ to Ready
        if (($op$ is load or store) & $S(op) = \text{cycle -1}$)
            add the ready successors of $op$ to Ready

Updating the Ready queue is the asymptotically expensive part of this algorithm

- If it is a problem, you can create a separate queue for each cycle
  - Add $op$ to queue for cycle when it will complete
  - Naïve implementation needs a lot of queues

- You can use even fewer queues
  - Number of queues bounded by maximum operation latency
  - Use them in a cyclic (or modulo) fashion

- Detailed algorithm is given at the end of this set of slides

I doubt that ready queue manipulation will be the big problem that you face.
Simple dependence graph from block report7.i, courtesy of lab3_ref.\textsuperscript{1}  

\textsuperscript{1} If you run the graph simplifier, the initial graph that lab3_ref builds for this block is much more complex, as it represents transitive dependences among I/O operations explicitly.
Lab 3

- Implement **one** scheduler for basic blocks in **ILOC**
- Choose your algorithm, choose your priorities, choose your tie-breakers
- Same **ILOC** subset as in lab 1, with addition of **NOP**
- Different execution model
  - Two asymmetric functional units
  - Latencies different from lab 1
  - Simulator different from lab 1

**Optional:** you may find it useful to have your scheduler output a Graphviz file for the dependence graph so that you can see what the scheduler is actually doing.

- The reference allocator will output a graphviz file, so you can look at it.
  - I found the graphviz output to be the best debugging tool available
- Installing graphviz on Mac OS 10 is challenging ... use **brew**
More List Scheduling

List scheduling algorithms fall into two distinct classes

Forward list scheduling
- Start with available operations
- Work forward in time
- Ready \( \Rightarrow \) all operands available

Backward list scheduling
- Start with no successors
- Work backward in time
- Ready \( \Rightarrow \) latency covers uses

Variations on list scheduling
- Prioritize critical path(s)
- Schedule last use as soon as possible
- Depth first in dependence graph
- Breadth first in dependence graph
- Prefer operation with most successors

Backward differs from forward in that:
1. Place predecessors on active queue, since they have individual latencies
2. Op only comes off the active queue onto the ready queue when all successors have been scheduled & their latencies covered.

See Gibbons & Muchnick, PLDI 1985, on Lectures page of class web site.
Local Scheduling

Forward and backward scheduling can produce different results

<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>

Block from SPEC benchmark “go”
### Example: Forward Schedule

#### Forward Schedule

<table>
<thead>
<tr>
<th></th>
<th>Int</th>
<th>Int</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>load(_1)</td>
<td>lshift</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>load(_2)</td>
<td>load(_3)</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>load(_4)</td>
<td>add(_1)</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>add(_2)</td>
<td>add(_3)</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>add(_4)</td>
<td>add(_1)</td>
<td>store(_1)</td>
</tr>
<tr>
<td>6</td>
<td>cmp</td>
<td></td>
<td>store(_2)</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td>store(_3)</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td>store(_4)</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td>store(_5)</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>cbr</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### ILOC Notation

- [ load\(_1\) ; lshift ; nop ]
- [ load\(_2\) ; load\(_3\) ; nop ]
- [ load\(_4\) ; add\(_1\) ; nop ]
- [ add\(_2\) ; add\(_3\) ; nop ]
- [ add\(_4\) ; add\(_1\) ; store\(_1\) ]
- [ cmp ; nop ; store\(_2\) ]
- [ nop ; nop ; store\(_3\) ]
- [ nop ; nop ; store\(_4\) ]
- [ nop ; nop ; store\(_5\) ]
- [ nop ; nop ; nop ]
- [ nop ; nop ; nop ]
- [ nop ; nop ; nop ]
- [ cbr ; nop ; nop ]

**Using “latency to root” as the priority function**

Assume 2 identical ALUs & 1 load/store unit
Example: Backward Schedule

Using “latency to root” as the priority function

<table>
<thead>
<tr>
<th></th>
<th>Int</th>
<th>Int</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>loadI₄</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>addI</td>
<td>lshift</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>add₄</td>
<td>loadI₃</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>add₃</td>
<td>loadI₂</td>
<td>store₅</td>
</tr>
<tr>
<td>5</td>
<td>add₂</td>
<td>loadI₁</td>
<td>store₄</td>
</tr>
<tr>
<td>6</td>
<td>add₁</td>
<td></td>
<td>store₃</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>store₂</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>store₁</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>cmp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>cbr</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Assume 2 identical ALUs & 1 load/store unit

Backward Schedule

ILOC Notation

1 fewer cycle in the backward schedule

COMP 412, Fall 2018
Instruction Scheduling

(What’s so hard?)

**Critical Points**

- An operation is **ready** to issue when its operands are available
- Multiple operations can be ready in any given cycle
  - *The scheduler must choose from among the ready operations*
- Moving operations can lengthen register lifetimes
- Placing uses near definitions can shorten register lifetimes
- In global scheduling, 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  (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 $(n_1, n_2) \in E$, $S(n_1) + delay(n_1) \leq S(n_2)$

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

$$L(S) = \max_{n \in N} (S(n) + delay(n))$$

The goal is to find the shortest possible correct schedule.

$S$ is time optimal if $L(S) \leq L(S_1)$, for all other schedules $S_1$

A schedule might also be optimal in terms of registers, energy consumption, or space....
Detailed Scheduling Algorithm I

**Idea:** Keep a collection of worklists $W[c]$, one per cycle

- We need $MaxC = \text{max delay} + 1$ such worklists
- Dependence graph is $(N,E)$

**Code:**

```plaintext
for each $n \in N$ do begin count[$n$] := 0; earliest[$n$] = 0 end
for each $(n1,n2) \in E$ do begin
    count[$n2$] := count[$n2$] + 1;
    successors[$n1$] := successors[$n1$] $\cup \{n2\};$
end
for $i := 0$ to $MaxC - 1$ do $W[i] := \emptyset;$
$Wcount := 0;$
for each $n \in N$ do
    if count[$n$] = 0 then begin
        $W[0] := W[0] \cup \{n\};$ $Wcount := Wcount + 1;$
    end
$c := 0;$ // $c$ is the cycle number
$cW := 0;$/ $cW$ is the number of the worklist for cycle $c$
$instr[c] := \emptyset;$
```

Successors & predecessors are reversed
Ken thought of them as running in the opposite direction
while \( W\text{count} > 0 \) do begin
    while \( W[cW] = \emptyset \) do begin
        \( c := c + 1; \) instr\([c]\) := \( \emptyset \); \( cW := \text{mod}(cW+1, \text{MaxC}); \)
    end
    nextc := \text{mod}(c+1, \text{MaxC});
    while \( W[cW] \neq \emptyset \) do begin
        select and remove an arbitrary instruction \( x \) from \( W[cW] \);
        if \( \exists \) free issue units of type\((x)\) on cycle \( c \) then begin
            instr\([c]\) := instr\([c]\) \( \cup \) \{\( x \)\}; \( W\text{count} := W\text{count} - 1; \)
            for each \( y \in \text{successors}[x] \) do begin
                count\([y]\) := count\([y]\) \(- 1; \)
                earliest\([y]\) := \max(earliest\([y]\), \( c+\text{delay}(x)\));
                if count\([y]\) = 0 then begin
                    loc := \text{mod}(earliest\([y]\), \text{MaxC});
                    W[loc] := W[loc] \( \cup \) \{\( y \)\}; \( W\text{count} := W\text{count} + 1; \)
                end
            end
        end
        else \( W[nextc] := W[nextc] \cup \{x\}; \)
    end
end

Successors & predecessors are reversed
Ken thought of them as running in the opposite direction