Instruction scheduling is an inherently machine dependent task. Unlike other code improvements which work with high- or medium-level intermediate representations, instruction scheduling works explicitly with machine instructions.

Instruction scheduling is often one of the last phases of code generation, performed after instruction selection and register allocation.

However, scheduling affects register allocation, so, ideally, we’d like to do instruction scheduling and register allocation simultaneously.

We’ll look at a scheduling algorithm that works directly on the generated machine code.
Modern processors use instruction pipelines to speed up execution **throughput** (total amount of work done in a given time).

The idea is that the execution of several instructions are overlapped in time, much the same as when a number of cars are assembled on an assembly line.

The execution of instructions is broken down in several (4–7) individual pipeline stages.

Pipelining doesn’t speed up the execution of individual instructions; rather, it increases the number of simultaneously executing instruction and the rate at which they are started and completed.

A typical 5-stage pipeline:

1. **Fetch**
   Fetch the instruction from memory (or instruction cache).

2. **Decode**
   Decode the instruction and fetch the values of registers that the instruction will need.

3. **Execute**
   Execute the instruction.

4. **Memory**
   Access memory (for load and store instructions).

5. **Write**
   Write the result of the instruction into the result register.

Not all instructions make use of all the pipeline stages. There is often more than one execute stage.

The MIPS R10000 has 7 stages: (1) Fetch, (2) Decode, (3) Issue Instruction and Read Registers, (4) Exec 1, (5) Exec 2, (6) Exec 3, (7) Register Write.
Pipeline Example I

<table>
<thead>
<tr>
<th>Clock Cycle</th>
<th>Without pipelining</th>
<th>With pipelining</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>lw $1,100($0)</td>
<td>lw $1,100($0)</td>
</tr>
<tr>
<td>2</td>
<td>Decode</td>
<td>Decode</td>
</tr>
<tr>
<td>3</td>
<td>Execute</td>
<td>Execute</td>
</tr>
<tr>
<td>4</td>
<td>Memory</td>
<td>Memory</td>
</tr>
<tr>
<td>5</td>
<td>Write</td>
<td>Write</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

lw $1,100($0) (This instruction doesn’t start until cycle 11)

Pipeline Example II

<table>
<thead>
<tr>
<th>Clock Cycle</th>
<th>Fetch</th>
<th>Decode</th>
<th>Execute</th>
<th>Memory</th>
<th>Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>lw $1,100($2)</td>
<td>Fetch</td>
<td>Decode</td>
<td>Execute</td>
<td>Memory</td>
</tr>
<tr>
<td>2</td>
<td>add $3,$4,$5</td>
<td>Fetch</td>
<td>Decode</td>
<td>Execute</td>
<td>Memory</td>
</tr>
<tr>
<td>3</td>
<td>sub $6,$7,$8</td>
<td>Fetch</td>
<td>Decode</td>
<td>Execute</td>
<td>Memory</td>
</tr>
<tr>
<td>4</td>
<td>mul $6,$7,55</td>
<td>Fetch</td>
<td>Decode</td>
<td>Execute</td>
<td>Memory</td>
</tr>
<tr>
<td>5</td>
<td>div $8,$9,66</td>
<td>Fetch</td>
<td>Decode</td>
<td>Execute</td>
<td>Memory</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>Fetch</td>
<td>Decode</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>Fetch</td>
<td>Decode</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Pipeline Data Hazards I

<table>
<thead>
<tr>
<th>Clock Cycle</th>
<th>Fetch</th>
<th>Decode</th>
<th>Execute</th>
<th>Memory</th>
<th>Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>lw $1,100($2)</td>
<td>Fetch</td>
<td>Decode</td>
<td>Execute</td>
<td>Memory</td>
</tr>
<tr>
<td>2</td>
<td>add $3,$1,44</td>
<td>Fetch</td>
<td>Decode</td>
<td>Execute</td>
<td>Memory</td>
</tr>
<tr>
<td>3</td>
<td>sub $6,$1,53</td>
<td>Fetch</td>
<td>Decode</td>
<td>Execute</td>
<td>Memory</td>
</tr>
<tr>
<td>4</td>
<td>mul $6,$7,55</td>
<td>Fetch</td>
<td>Decode</td>
<td>Execute</td>
<td>Memory</td>
</tr>
<tr>
<td>5</td>
<td>div $8,$9,66</td>
<td>Fetch</td>
<td>Decode</td>
<td>Execute</td>
<td>Memory</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>Fetch</td>
<td>Decode</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>Fetch</td>
<td>Decode</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The previous example shows one kind of problem we have with pipelines. Data hazards occur when the result of one instruction isn’t ready in time for when that result is needed.

Some processors will detect such situations and stall the pipeline until the value is ready. This is called a hardware/pipeline interlock.

Interlocks are expensive (extra control logic on the chip which takes up valuable chip real estate) so some processors do away with them. Instead they rely on compilers and assemblers to insert NOPs when needed.

Example: On a MIPS 2000, the value loaded by a ld instruction isn’t available to the immediately following instruction. The processor doesn’t have hardware interlocks. If you give the following code the the assembler

```
    ld 100($4), $5
    add $5, $5, 56
```

it will insert the necessary NOPs:

```
    ld 100($4), $5
    nop
    add $5, $5, 56
```

Even if the hardware does have interlocks, the compiler (or assembler) should pay attention to the order in which instructions are scheduled. A well scheduled program may be up to 50% faster than an naïvely scheduled one.

Some processors take care of data hazards themselves – the pipeline is stalled for a number of cycles until the hazard is resolved. This is like inserting one or more bubbles (NOPs) in the pipeline.

The compiler can sometimes reorder instructions to avoid stalls.
Dependency Graphs

An instruction scheduler reorders the instructions in an attempt at minimizing pipeline stalls.

Obviously, we must not rearrange the code so that (when executed) it produces a different result from before!

\[
\begin{align*}
\text{Wrong!} & \quad \text{li } 2, 66 \\
\rightarrow & \quad \text{add } 3, 2, 44
\end{align*}
\]

We must therefore model all the dependencies between the instructions. We store this information in a directed dependency graph. The nodes of the graph are the instructions we’re scheduling, and there’s an edge \( a \Rightarrow b \) if instruction \( a \) must come before instruction \( b \).

There can be three kinds of dependencies between instructions:

- **Flow dependence**
- **True dependence** or **Definition-use dependence**
- **Anti-dependence** or **Use-definition dependence**

**Flow dependence**

\[
\begin{align*}
(i) & \quad X := \cdots \\
(j) & \quad \cdots := X
\end{align*}
\]

Instruction \((i)\) generates (defines) a value which is used by instruction \((j)\). We write \((i) \rightarrow (j)\).

**True dependence**

\[
\begin{align*}
(i) & \quad \cdots := X \\
(j) & \quad \cdots := X
\end{align*}
\]

**Anti-dependence**

\[
\begin{align*}
(i) & \quad \cdots := X \\
(j) & \quad X := \cdots
\end{align*}
\]

Instruction \((i)\) uses a value overwritten by instruction \((j)\). We write \((i) \rightarrow+ (j)\).
Output-dependence

Also, definition-definition dependence.

(i) \( X := \cdots \)
(j) \( X := \cdots \)

Instructions (i) and (j) both assign to \((\text{define})\) the same variable. We write \((i) \rightarrow (j)\).

Regardless of the type of dependence, if instruction \((j)\) depends on \((i)\), then \((i)\) has to be scheduled before \((j)\).

Flow Dependence Example

(1) \textit{ld} \ 100($3), \ $2
    \( \cdots \) \( \Rightarrow \) (1) \( \rightarrow \) (5)
(5) \textit{add} \ $5, \ $2, \ 55

- We can’t move (1) after (5). If we did, \$2 would not have the correct value at (5). Hence there is a flow dependence between (1) and (5).

Anti-Dependence Example

(2) \textit{add} \ $5, \ $2, \ 55
    \( \cdots \) \( \Rightarrow \) (2) \( \rightarrow \) (6)
(6) \textit{li} \ $2, \ 33

- \$2 gets a new value in (6). If we moved (2) after (6) \$2 would have the wrong value in (6).

Output-Dependence Example

(3) \textit{li} \ $2, \ 33
(4) \textit{add} \ $2, \ $5, \ 55 \( \Rightarrow \) (3) \( \rightarrow \) (4)
(5) \textit{add} \ $6, \ $2, \ 44

- If we switched (3) and (4), \$2 would get the wrong value in (3).

Complex Example

(1) \textit{lw} \ $1, \ 100($3)
(2) \textit{lw} \ $2, \ 200($3)
(3) \textit{add} \ $3, \ $3, \ 22
(4) \textit{sub} \ $1, \ $1, \ $3
(5) \textit{sw} \ $3, \ 500($0)
(6) \textit{add} \ $2, \ $2, \ 11
(7) \textit{mul} \ $4, \ $3, \ 44
(8) \textit{mul} \ $5, \ $4, \ $1
(9) \textit{div} \ $5, \ $4, \ $3
If instruction (b) depends on instruction (a) then (b) can be scheduled after a delay of one cycle.

Real pipelines are more complex than this....

Topological Sorting

- In the following example, candidates are marked with a c.
- When there is more than one candidate to choose from, we pick one at random.
- There are often many possible topological orders to choose from. For example, since the watch node has no dependencies at all, I can put it on at any time.

Topological Sorting I

- Problem: How to get dressed? I can't put my clothes on in an arbitrary order: socks have to be put on before shoes, shirt before tie, and so on.
- Each garment becomes a node in a DAG, and there's an edge from $u$ to $v$ if I have to put on $u$ before $v$.

---

Topological Sorting:

"Order the nodes of the graph $G$ in a sequence such that if $x \rightarrow y$ is an edge is $G$ then $x$ comes before $y$ in the ordering."

Simple Algorithm:

Repeat until no more nodes:
- Pick a node $n$ without predecessors.
- Print $n$.
- Delete $n$ and all its outgoing edges.
Last := {}; /* Last scheduled instr. */
REPEAT
  Candidates := set of all nodes without predecessors (incoming edges);
  Colliding := set of all instructions that collide with the instructions in Last; i.e. nodes \( n \) for which there is an arch Last \( \rightarrow n \).
  realCandidates := Candidates - Colliding;
  IF realCandidates \( \neq \{\} \) THEN
    b := Use heuristic to pick the best realCandidate to schedule;
    Remove b and all its outgoing edges from the graph;
    print b; Last := \{b\};
  ELSE
    print NOP; Last := \{\};
  ENDIF
UNTIL Candidates = {};

Heuristic

- In the "standard" topological sorting algorithm we pick a node at random when there are several candidate nodes to choose from. In our scheduling algorithm we need a better heuristic.
- If there is more than one real candidate (a candidate that doesn't collide with the last instruction scheduled) we choose one according to these criteria:
  1. Pick the candidate with the largest number of outgoing edges. Scheduling this instruction early will hopefully give us more freedom to schedule the remaining instructions.
  2. If all candidates have the same number of outgoing edges, pick the one that has the longest path to a leaf.

Example

Example I (a)

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>lw</td>
<td>$1, 100($3)</td>
</tr>
<tr>
<td>2</td>
<td>lw</td>
<td>$2, 200($3)</td>
</tr>
<tr>
<td>3</td>
<td>add</td>
<td>$3, $3, 22</td>
</tr>
<tr>
<td>4</td>
<td>sub</td>
<td>$1, $1, $3</td>
</tr>
<tr>
<td>5</td>
<td>sw</td>
<td>$3, 500($0)</td>
</tr>
<tr>
<td>6</td>
<td>add</td>
<td>$2, $2, 11</td>
</tr>
<tr>
<td>7</td>
<td>mul</td>
<td>$4, $3, 44</td>
</tr>
<tr>
<td>8</td>
<td>mul</td>
<td>$5, $4, $1</td>
</tr>
<tr>
<td>9</td>
<td>div</td>
<td>$5, $4, $3</td>
</tr>
</tbody>
</table>
If instruction \((b)\) depends on instruction \((a)\) then \((b)\) can be scheduled after a delay of one cycle.

- We mark real candidates with an \(r\).

### Example I (b)

- If instruction \((b)\) depends on instruction \((a)\) then \((b)\) can be scheduled after a delay of one cycle.
- We mark real candidates with an \(r\).

### Example I (c)

- If instruction \((b)\) depends on instruction \((a)\) then \((b)\) can be scheduled after a delay of one cycle.
- We mark real candidates with an \(r\).

### Example I (d)

- If instruction \((b)\) depends on instruction \((a)\) then \((b)\) can be scheduled after a delay of one cycle.
- We mark real candidates with an \(r\).

### Example I (e)

- If instruction \((b)\) depends on instruction \((a)\) then \((b)\) can be scheduled after a delay of one cycle.
- We mark real candidates with an \(r\).
Some processors have instructions with pipeline hot-spots: A value that is computed in one cycle must be consumed in the next cycle (before it is overwritten).

We have assumed that the delay between two instructions is always exactly 1 (one) cycle. Real processors require different delays depending on the instruction.
Homework I

Consider the following basic block:

1. lw  $1, 100($2)
2. li  $3, 100
3. add $2, $3, 55
4. sub $1, $3, 66
5. li  $5, 99
6. add $7, $3, $5
7. li  $3, 93
8. add $9, $2, $7
9. sub $1, $9, $3

1. Construct the dependency graph.
2. Apply the scheduling algorithm to the graph. Assume that there is a one-cycle delay between an instruction and when it's computed value can be used.
3. Is the generated schedule an optimal one?

Exam Problem I (415.730 '96)

1. What is a pipeline data hazard?
2. Why is it important for the compiler to perform aggressive instruction scheduling, even when the architecture has hardware interlocks?
3. Describe the algorithms and data structures needed to perform instruction scheduling in the presence of pipeline data hazards.