Pipelining

load a, r1
load b, r2
add r1, r2, r3
store r3, c
**Data hazard:** instruction uses result of earlier instruction

\[
\begin{align*}
\text{r1} &= \text{load r4} \\
\text{r3} &= \text{r1} + \text{r2}
\end{align*}
\]

**Control hazard:** instruction execution conditional on result of earlier instruction

\[
\begin{align*}
\text{if r1 goto label} \\
\text{r4} &= \text{r2} + \text{r3}
\end{align*}
\]

**Structural hazard:** instruction requires resources in use by earlier instruction

\[
\begin{align*}
\text{r1} &= \text{r2} \times \text{r3} \\
\text{r4} &= \text{r5} \times \text{r6}
\end{align*}
\]
Instruction Scheduling

instruction sequence

instruction scheduler

same set of instructions
different order
same semantics
An instruction $j$ occurring after $i$ is dependent on $i$ if moving $j$ before $i$ would change the semantics of the program.

- **Control dependence:** $i$ determines whether $j$ executes

- **Data dependences:**
  - True dependence (WR):
    - $i$ computes value, $j$ uses it
    - $i: x = \ldots$
    - $j: \ldots = x$
  - Output dependence (WW):
    - $i$ writes value, $j$ overwrites it
    - $i: x = \ldots$
    - $j: x = \ldots$
  - Anti-dependence (RW):
    - $i$ reads value, $j$ overwrites it
    - $i: \ldots = x$
    - $j: x = \ldots$
  - Input dependence (RR):
    - $i$ reads value, $j$ reads it too
    - $i: \ldots = x$
    - $j: \ldots = x$
Dependence Graph

- each instruction is a node
- edge $i \rightarrow j$ if $j$ depends on $i$

r1 = A
r2 = B
r1 = r1 + r2
A = r1
r1 = C
r1 = C
r2 = D
r2 = r1 + r2
each instruction is a node
edge $i \rightarrow j$ if $j$ depends on $i$

\[
\begin{align*}
    r_1 &= A \\
    r_2 &= B \\
    r_1 &= r_1 + r_2 \\
    A &= r_1 \\
    r_1 &= C \\
    r_2 &= D \\
    r_2 &= r_1 + r_2
\end{align*}
\]
Dependence Graph

- each instruction is a node
- edge $i \rightarrow j$ if $j$ depends on $i$

\[
\begin{align*}
    r1 &= A \\
    r2 &= B \\
    r1 &= r1 + r2 \\
    A &= r1 \\
    r1 &= C \\
    r2 &= D \\
    r2 &= r1 + r2
\end{align*}
\]
Dependence Graph

- each instruction is a node
- edge $i \rightarrow j$ if $j$ depends on $i$

\begin{align*}
  r_1 &= A \\
r_2 &= B \\
r_3 &= r_1 + r_2 \\
A &= r_3 \\
r_4 &= C \\
r_5 &= D \\
r_6 &= r_4 + r_5
\end{align*}
Dependence Graph

- each instruction is a node
- edge $i \rightarrow j$ if $j$ depends on $i$

\[ \begin{align*}
    r_1 &= A \\
    r_2 &= B \\
    \text{stall} \\
    r_3 &= r_1 + r_2 \\
    A &= r_3 \\
    r_4 &= C \\
    r_5 &= D \\
    \text{stall} \\
    r_6 &= r_4 + r_5
\end{align*} \]
Dependence Graph

- each instruction is a node
- edge $i \rightarrow j$ if $j$ depends on $i$

```
r1 = A
r2 = B
r4 = C
r5 = D
r3 = r1 + r2
A = r3
r6 = r4 + r5
r1 = A r2 = B r4 = C r5 = D
  \2  \2  \2  \2
r3 = r1 + r2 r6 = r4 + r5
  \1
A = r3
```
Algorithm Schedule():
1: assign each instruction a priority (heuristic)
2: create a list (priority queue) of eligible instructions
   - all predecessors already scheduled
   - all delays already satisfied
3: repeat
4: remove highest-priority instruction from list
5: add it to schedule
6: add newly-eligible instructions to list
7: until all instructions have been scheduled
Priority Heuristics

- longest (latency-weighted) path from \( n \) to a sink node
- number of immediate successors
- number of descendants (not necessarily immediate)
- latency of \( n \)
- increase priority for last use of a value
- ...
Register allocation may introduce dependences

\[
\begin{align*}
a &= b + c \\
e &= d + f \\
r1 &= r2 + r3 \\
r2 &= r4 + r5
\end{align*}
\]

Also, any spills need to be scheduled.

Instruction scheduling may increase register pressure

\[
\begin{align*}
a &= \ldots \\
\ldots &= a \\
a &= \ldots \\
\text{other instructions} \\
\ldots &= a
\end{align*}
\]

Typical solution: schedule, allocate registers, schedule again
Software Pipelining

```c
for(i=0; i<100; i++) {
    r1 = a[i];
    r2 = b[i];
    r3 = r1 + r2;
    c[i] = r3;
}
```

```c
r1 = a[0];
r2 = b[0];
for(i=0; i<99; i++) {
    r4 = a[i+1];
    r5 = b[i+1];
    r3 = r1 + r2;
    c[i] = r3;
    r1 = r4;
    r2 = r5;
}
r3 = r1 + r2;
c[99] = r3;
```
Software Pipelining

```c
for(i=0; i<100; i++) {
    r1 = a[i];
    r2 = b[i];
    r3 = r1 + r2;
    c[i] = r3;
}

r1 = a[0];
r2 = b[0];
for(i=0; i<99; i++) {
    r4 = a[i+1];
    r5 = b[i+1];
    r3 = r1 + r2;
    c[i] = r3;
    r1 = r4;
    r2 = r5;
}

r3 = r1 + r2;
c[99] = r3;
```
Example CPU: 2 integer, 2 floating point, and 1 branch instruction per cycle.

In each time slot, schedule the best 2 integer, the best 2 fp, and the best branch instruction from the list.
Many processors support out-of-order execution.

**Advantages of OOE**
- dependencies and latency may not be statically known
- code may run on different hardware
- hardware register renaming

**Advantages of Static Instruction Scheduling**
- OOE limited to short dependence paths
- compiler has more time to find good schedule
- hardware resource constraints

For best performance, combine both.