Software Pipelining

- Software pipelining is an IS technique that reorders the instructions in a loop.
  - Possibly moving instructions from one iteration to the previous or the next iteration.
  - Very large improvements in running time are possible.
- The first serious approach to software pipelining was presented by Aiken & Nicolau.
  - Impractical as it ignores resource hazards (focusing only on data-dependence constraints).
  - But sparked a large amount of follow-on research.

Goal of SP

- Increase distance between dependent operations by moving destination operation to a later iteration

A: \( a \leftarrow \text{ld}[d] \)
B: \( b \leftarrow a \times a \)
C: \( \text{st}[d], b \)
D: \( d \leftarrow d + 4 \)

Assume all have latency of 2

Can we decrease the latency?

- Lets unroll

\[
\begin{align*}
A: & \quad a \leftarrow \text{ld}[d] \\
B: & \quad b \leftarrow a \times a \\
C: & \quad \text{st}[d], b \\
D: & \quad d \leftarrow d + 4 \\
A1: & \quad a \leftarrow \text{ld}[d] \\
B1: & \quad b \leftarrow a \times a \\
C1: & \quad \text{st}[d], b \\
D1: & \quad d \leftarrow d + 4
\end{align*}
\]
Rename variables

A: \( a \leftarrow \text{ld} [d] \)
B: \( b \leftarrow a \ast a \)
C: \( \text{st} [d], b \)
D: \( d1 \leftarrow d + 4 \)
A1: \( a1 \leftarrow \text{ld} [d1] \)
B1: \( b1 \leftarrow a1 \ast a1 \)
C1: \( \text{st} [d1], b1 \)
D1: \( d \leftarrow d1 + 4 \)

Schedule

A: \( a \leftarrow \text{ld} [d] \)
B: \( b \leftarrow a \ast a \)
C: \( \text{st} [d], b \)
D: \( d1 \leftarrow d + 4 \)
A1: \( a1 \leftarrow \text{ld} [d1] \)
B1: \( b1 \leftarrow a1 \ast a1 \)
C1: \( \text{st} [d1], b1 \)
D1: \( d \leftarrow d1 + 4 \)

Unroll Some More

A: \( a \leftarrow \text{ld} [d] \)
B: \( b \leftarrow a \ast a \)
C: \( \text{st} [d], b \)
D: \( d1 \leftarrow d + 4 \)
A1: \( a1 \leftarrow \text{ld} [d1] \)
B1: \( b1 \leftarrow a1 \ast a1 \)
C1: \( \text{st} [d1], b1 \)
D1: \( d2 \leftarrow d1 + 4 \)
A2: \( a2 \leftarrow \text{ld} [d2] \)
B2: \( b2 \leftarrow a2 \ast a2 \)
C2: \( \text{st} [d2], b2 \)
D2: \( d \leftarrow d2 + 4 \)

Unroll Some More

A: \( a \leftarrow \text{ld} [d] \)
B: \( b \leftarrow a \ast a \)
C: \( \text{st} [d], b \)
D: \( d1 \leftarrow d + 4 \)
A1: \( a1 \leftarrow \text{ld} [d1] \)
B1: \( b1 \leftarrow a1 \ast a1 \)
C1: \( \text{st} [d1], b1 \)
D1: \( d2 \leftarrow d1 + 4 \)
A2: \( a2 \leftarrow \text{ld} [d2] \)
B2: \( b2 \leftarrow a2 \ast a2 \)
C2: \( \text{st} [d2], b2 \)
D2: \( d \leftarrow d2 + 4 \)
One More Time

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>A</td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>A1</td>
<td>B1</td>
<td>C1</td>
<td>D1</td>
</tr>
<tr>
<td>B</td>
<td>C</td>
<td>D</td>
<td>A</td>
</tr>
</tbody>
</table>

Can Rearrange

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>A</td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>A1</td>
<td>B1</td>
<td>C1</td>
<td>D1</td>
</tr>
<tr>
<td>B</td>
<td>C</td>
<td>D</td>
<td>A</td>
</tr>
</tbody>
</table>

Rearrange

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>A</td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>A1</td>
<td>B1</td>
<td>C1</td>
<td>D1</td>
</tr>
<tr>
<td>B</td>
<td>C</td>
<td>D</td>
<td>A</td>
</tr>
</tbody>
</table>

Rearrange

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>A</td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>A1</td>
<td>B1</td>
<td>C1</td>
<td>D1</td>
</tr>
<tr>
<td>B</td>
<td>C</td>
<td>D</td>
<td>A</td>
</tr>
</tbody>
</table>

© Seth Copen Goldstein 2000-1
Goal of SP

- Increase distance between dependent operations by moving destination operation to a later iteration
- But also, to uncover ILP across iteration boundaries!

Example

Assume operating on a infinite wide machine
Example

Assume operating on an infinite wide machine

```
Prolog
```

```
loop body
```

```
epilog
```

Dealing with exit conditions

```
for (i=0; i<N; i++)
{
    A_i = B_{i-1} + C_{i-2}
    if (i >= N) goto done
    if (i+1 == N) goto last
    i = 1
    A_1 = B_0 + C_i
    if (i+2 == N) goto epilog
    i = 2
    loop:
    A_i = B_{i-1} + C_{i-2}
    i++
    if (i < N) goto loop
    epilog:
    B_i = C_{i-1}
    last:
    C_i = done:
```

Loop Unrolling V. SP

For SuperScalar or VLIW
- Loop Unrolling reduces loop overhead
- Software Pipelining reduces fill/drain
- Best is if you combine them

Aiken/Nicolau Scheduling

Step 1

Perform scalar replacement to eliminate memory references where possible.

```
for i:=1 to N do
    a := j @ V[i-1]
    b := a @ f
    c := e @ j
    d := f @ c
    e := b @ d
    f := U[i]
    g := V[i] := b
    h := W[i] := d
    j := X[i]
```

```
for i:=1 to N do
    a := j @ b
    b := a @ f
    c := e @ j
    d := f @ c
    e := b @ d
    f := U[i]
    g := V[i] := b
    h := W[i] := d
    j := X[i]
```
Aiken/Nicolau Scheduling
Step 2

Unroll the loop and compute the data-dependence graph (DDG).

DDG for rolled loop:
for i:=1 to N do
  a := j ⊕ b
  b := a ⊕ f
  c := e ⊕ j
  d := f ⊕ c
  e := b ⊕ d
  f := U[i]
  g: V[i] := b
  h: W[i] := d
  j := X[i]

Aiken/Nicolau Scheduling
Step 2, cont’d

DDG for unrolled loop:
for i:=1 to N do
  a := j ⊕ b
  b := a ⊕ f
  c := e ⊕ j
  d := f ⊕ c
  e := b ⊕ d
  f := U[i]
  g: V[i] := b
  h: W[i] := d
  j := X[i]

basically, you’re emulating a superscalar with infinite resources, infinite register renaming, always predicting the loop-back branch: thus, just pure data dependency.
Aiken/Nicolau Scheduling

Step 4

Find repeating patterns of instructions.

<table>
<thead>
<tr>
<th>Iteration</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cycle</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>cfj fj fj fj fj fj</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>bd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>kgh a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>cb</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>dg a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>ah b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>cg a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>db</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>ah g a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>cb</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>dg a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>ah b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>cg</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>d</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>ah</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

iteration

1    2   3   4   5   6

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Go back and relate slopes to DDG

Aiken/Nicolau Scheduling

Step 5

"Coalesce" the slopes.

<table>
<thead>
<tr>
<th>Iteration</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cycle</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>cfj fj fj fj fj fj</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>bd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>kgh a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>cb</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>dg a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>ah b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>cg g a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>d b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>ah g a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>cb</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>dg a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>ah b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>cg</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>d</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>ah</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

iteration

1 2 3 4 5 6

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Go back and relate slopes to DDG
Aiken/Nicolau Scheduling

Step 6

Find the loop body and "reroll" the loop.

<table>
<thead>
<tr>
<th>iteration</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>f</td>
<td>j</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>d</td>
<td>f</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td></td>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>g</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Step 7

Generate code.

(Assume VLIW-like machine for this example. The instructions on each line should be issued in parallel.)

L:
di := f, ci := c, bi-1 := a, fi-1 := f
b0 := b0 @ b1, Wi := d0, Vi+1 := b1, fi := U[i+2], j := X[i+2]
ci := c, bi := b0 @ b1, Vi := bi, bi := i := i+2, if i < N-2 goto L

da0 := f0 @ f1, h0 := a0 @ f0
a0 := b0 @ b1, Wi := h0
b0 := b0 @ b1, Wi := h0

Aiken/Nicolau Scheduling

Step 8

• Since several versions of a variable (e.g., j_i and j_{i+1}) might be live simultaneously, we need to add new temps and moves

L:
di := f, ci := c, bi-1 := a, fi-1 := f
b0 := b0 @ b1, Wi := d0, Vi+1 := b1, fi := U[i+2], j := X[i+2]
ci := c, bi := b0 @ b1, Vi := bi, bi := i := i+2, if i < N-2 goto L

da0 := f0 @ f1, h0 := a0 @ f0
a0 := b0 @ b1, Wi := h0
b0 := b0 @ b1, Wi := h0

Aiken/Nicolau Scheduling

Step 8

• Since several versions of a variable (e.g., j_i and j_{i+1}) might be live simultaneously, we need to add new temps and moves
Aiken/Nicolau Scheduling

Step 8

- Since several versions of a variable (e.g., \(j_i\) and \(j_{i+1}\)) might be live simultaneously, we need to add new temps and moves

\[
\begin{align*}
a_1 &= a_0 \oplus b_0 & c_1 &= e_0 \oplus j_0 & f_1 &= U[1] & j_1 &= X[1] \\
b_1 &= a_1 \oplus f_0 & d_1 &= c_0 \oplus c_1 & e_1 &= U[1] \oplus j_1 & j_2 &= X[2] \\
c_2 &= e_1 \oplus j_1 & f_0 &= f_0 \oplus c_0 & f_'' &= U[2] & j_3 &= X[3] \\
d_2 &= f_1 \oplus c_1 & e_2 &= b_1 \oplus d_1 & f'' &= U[3] & j' &= X[4] \\
e_3 &= b_1 \oplus d_1 & f_1 &= f_1 \oplus c_1 & f_4 &= U[4] & j_4 &= X[4] \\
\end{align*}
\]

L:

\[
\begin{align*}
d_i &= e_i \oplus c_i & b_{i+1} &= a_i \oplus b_i & b_i &= a_i \oplus f_i \oplus c_i & f_i &= U[i+2] & j_i &= X[i+2] \\
e_{i+1} &= b_i \oplus d_i & W[i] &= d_i & j_{i+1} &= X[i+1] & j_{i+2} &= X[i+2] \\
e_i &= b_i \oplus d_i & W[i+1] &= d_i & j_{i+1} &= X[i+1] & j_{i+2} &= X[i+2] \\
\end{align*}
\]

i := 3

\[
\begin{align*}
d_{N-1} &= f_{N-2} \oplus c_{N-1} & b_N &= a_N \oplus f_{N-1} & W[N] &= d_N \\
e_{N-1} &= b_N \oplus d_N & b_N &= a_N \oplus f_{N-1} & W[N-1] &= d_N \\
d_N &= f_{N-2} \oplus c_{N-1} & e_N &= b_N \oplus d_N & W[N-2] &= d_N \\
\end{align*}
\]

Next Step in SP

- AN88 did not deal with resource constraints.
- Modulo Scheduling is a SP algorithm that does.
- It schedules the loop based on
  - resource constraints
  - precedence constraints
- Basically, it's list scheduling that takes into account resource conflicts from overlapping iterations

Resource Constraints

- Minimally indivisible sequences, \(i\) and \(j\), can execute together if combined resources in a step do not exceed available resources.
- \(R(i)\) is a resource configuration vector
  \(R(i)\) is the number of units of resource \(i\)
- \(r(i)\) is a resource usage vector s.t.
  \(0 \leq r(i) \leq R(i)\)
- Each node in \(G\) has an associated \(r(i)\)

Software Pipelining Goal

- Find the same schedule for each iteration.
- Stagger by iteration initiation interval, \(s\)
- Goal: minimize \(s\)
Software Pipelining Goal

- Find the same schedule for each iteration.
- Stagger by iteration initiation interval, $s$
- Goal: minimize $s$.

resources must be within constraints
Software Pipelining Goal

- Find the same schedule for each iteration.
- Stagger by iteration initiation interval, $s$
- Goal: minimize $s$.

resources must be within constraints

Precedence Constraints

- Review: for acyclic scheduling, constraint is just the required delay between two ops $u, v$: $\langle d(u,v) \rangle$
- For an edge, $u \rightarrow v$, we must have $\sigma(v) - \sigma(u) \geq d(u,v)$

Software Pipelining Goal

- Find the same schedule for each iteration.
- Stagger by iteration initiation interval, $s$
- Goal: minimize $s$.

resources must be within constraints

Precedence Constraints

- Cyclic: constraint becomes a tuple: $\langle p, d \rangle$
  - $p$ is the minimum iteration delay (or the loop carried dependence distance)
  - $d$ is the delay
- For an edge, $u \rightarrow v$, we must have $\sigma(v) - \sigma(u) \geq d(u,v) - s*p(u,v)$
- $p \geq 0$
- If data dependence is
  - within an iteration, $p=0$
  - loop-carried across $p$ iter boundaries, $p>0$
Iterative Approach

• Finding minimum $S$ that satisfies the constraints is NP-Complete.
• Heuristic:
  - Find lower and upper bounds for $S$
  - foreach $s$ from lower to upper bound?
    • Schedule graph.
    • If succeed, done
    • Otherwise try again (with next higher $s$)
• Thus: “Iterative Modulo Scheduling” Rau, et.al.

Lower Bounds

• Resource Constraints: $S_R$ (also called $\text{II}_{\text{res}}$)
  maximum over all resources of # of uses divided by # available... rounded up or down?

• Precedence Constraints: $S_E$ (also called $\text{II}_{\text{rec}}$)
  max over all cycles: $d(c)/p(c)$

In practice, one is easy, other is hard.
Tim’s secret approach: just use $S_R$ as lower bound, then do binary search for best $S$

Iterative Approach

• Heuristic:
  - Find lower and upper bounds for $S$
  - foreach $s$ from lower to upper bound?
    • Schedule graph.
    • If succeed, done
    • Otherwise try again (with next higher $s$)

So the key difference:
- AN88 does not assume $S$ when scheduling
- IMS must assume an $S$ for each scheduling attempt to understand resource conflicts

Acyclic Example

Lower Bound: $S_R=2$
Upper Bound: 5
Lower Bound on $s$

- Assume 1 ALU and 1 MU
- Assume latency Op or load is 1 cycle

```
for i:=1 to N do
    a := j ⊕ b
    b := a ⊕ f
    c := e ⊕ j
    d := f ⊕ c
    e := b ⊕ d
    f := U[i]
    g := V[i] := b
    h := W[i] := d
    j := X[i]
```

Resources => 5 cycles
Dependencies => 3 cycles

Scheduling data structures

To schedule for initiation interval $s$:
- Create a resource table with $s$ rows and $R$ columns
- Create a vector, $\sigma$, of length $N$ for $n$ instructions in the loop
  - $\sigma[n] =$ the time at which $n$ is scheduled, or NONE
- Prioritize instructions by some heuristic
  - critical path (or cycle)
  - resource critical

Scheduling algorithm

- Pick an instruction, $n$
- Calculate earliest time due to dependence constraints
  For all $x$=pred($n$),
    earliest = max(earliest, $\sigma(x)+d(x,n)-s\cdot p(x,n)$)
- try and schedule $n$ from earliest to (earliest+$s$-1)
  s.t. resource constraints are obeyed.
  - possible twist: deschedule a conflicting node to make
t    way for $n$, maybe randomly, like sim anneal
- If we fail, then this schedule is faulty
  (i.e. give up on this $s$)

Scheduling algorithm – cont.

- We now schedule $n$ at earliest, i.e., $\sigma(n) =$ earliest
- Fix up schedule
  - Successors, $x$, of $n$ must be scheduled s.t.
    $\sigma(x) =$ $\sigma(n)+d(n,x)-s\cdot p(n,x)$, otherwise they
    are removed (descheduled) and put back on worklist.
  - repeat this some number of times until either
    - succeed, then register allocate
    - fail, then increase $s$
Simplest Example

```plaintext
for () {
    a = b+c
    b = a*a
    c = a*194
}
```

What is IIres?
What is IIrec?

Simplest Example

```plaintext
for () {
    a = b+c
    b = a*a
    c = a*194
}
```

Try II = 2

Modulo Resource Table:

- 0
- 1
- 1
- 1

Simplest Example

```plaintext
for () {
    a = b+c
    b = a*a
    c = a*194
}
```

Try II = 2

Modulo Resource Table:

- 0
- 1
- 1
- 1
### Simplest Example

```cpp
for () {
    a = b+c
    b = a*a
    c = a*194
}
```

Try II = 2

**Modulo Resource Table:**

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```
earliest a: sigma(c) + delay(c) - 2 = 2+1-2 = 1
```

### Example

```cpp
for i:=1 to N do
    a := j ⊕ b
    b := a ⊕ f
    c := e ⊕ j
    d := f ⊕ c
    e := b ⊕ d
    f := U[i]
    g: V[i] := b
    h: W[i] := d
    j := X[i]
```

Priorities: ?

```
<1,1> | <1,1> | <1,1>
<0,1> | <0,1> | <0,1>
<1,1> | <0,1> | <0,1> | <0,1>
```

Lesson: lower bound may not be achievable
Example

for i:=1 to N do
a := j ⊕ b
b := a ⊕ f
c := e ⊕ j
d := f ⊕ c
e := b ⊕ d
f := U[i]
g := V[i] := b
h := W[i] := d
j := X[i]

Priorities: c,d,e,a,b,f,j,g,h

\[
\begin{align*}
a &<1,1> \\
b &<0,1> \\
c &<1,1> \\
d &<1,1> \\
e &<0,1> \\
f &<0,1> \\
g &<1,1> \\
h &<0,1> \\
j &<0,1> \\
\end{align*}
\]

for i:=1 to N do
a := j ⊕ b
b := a ⊕ f
c := e ⊕ j
d := f ⊕ c
e := b ⊕ d
f := U[i]
g := V[i] := b
h := W[i] := d
j := X[i]

Priorities: c,d,e,a,b,f,j,g,h

\[
\begin{align*}
a &<0,1> \\
b &<1,1> \\
c &<0,1> \\
d &<1,1> \\
e &<0,1> \\
f &<0,1> \\
g &<1,1> \\
h &<0,1> \\
j &<0,1> \\
\end{align*}
\]
for i:=1 to N do
  a := j ⊕ b
  b := a ⊕ f
  c := e ⊕ j
  d := f ⊕ c
  e := b ⊕ d
  f := U[i]
  g: V[i] := b
  h: W[i] := d
  j := X[i]

Priorities: b,f,g,h

for i:=1 to N do
  a := j ⊕ b
  b := a ⊕ f
  c := e ⊕ j
  d := f ⊕ c
  e := b ⊕ d
  f := U[i]
  g: V[i] := b
  h: W[i] := d
  j := X[i]

Priorities: e,f,g,h

b causes b→e edge violation

e causes e→c edge violation
Creating the Loop

- Create the body from the schedule.
- Determine which iteration an instruction falls into
  - Mark its sources and dest as belonging to that iteration.
  - Add Moves to update registers
- Prolog fills in gaps at beginning
  - For each move we will have an instruction in prolog, and we fill in dependent instructions
- Epilog fills in gaps at end

for i := 1 to N do
  a := j @ b
  b := a @ f
  c := e @ g
  d := f @ c
  e := b @ d
  f := U[i]
g: V[i] := b
h: W[i] := d
j := X[i]

Priorities: j, g, h

s = 5

for i := 1 to N do
  a := j @ b
  b := a @ f
  c := e @ g
  d := f @ c
  e := b @ d
  f := U[i]
g: V[i] := b
h: W[i] := d
j := X[i]

Priorities: g, h

s = 5
Conditionals

- What about internal control structure, i.e., conditionals
- Three approaches
  - Schedule both sides and use conditional moves
  - Schedule each side, then make the body of the conditional a macro op with appropriate resource vector
  - Trace schedule the loop

What to take away

- Dependence analysis is very important
- Software pipelining is cool
- Registers are a key resource