Instruction-level Parallelism

- Most modern processors have the ability to execute several adjacent instructions simultaneously.
  - Pipelined machines.
  - Very-long-instruction-word machines (VLIW).
  - Superscalar machines.
  - Dynamic scheduling/out-of-order machines.
- ILP is limited by several kinds of *execution constraints*:
  - Data dependence constraints.
  - Resource constraints ("hazards")
  - Control hazards

Execution Constraints

- Data-dependence constraints:
  - If instruction A computes a value that is read by instruction B, then B cannot execute before A is completed.
- Resource hazards:
  - Limited # of functional units:
    - If there are *n* functional units (e.g., multipliers), then only *n* instructions of the same kind can execute at once.
  - Limited instruction issue:
    - If the instruction-issue unit can issue only *n* instructions at a time, then this limits ILP.
  - Limited register set:
    - Any schedule of instructions must have a valid register allocation.

Instruction Scheduling

- The purpose of instruction scheduling (IS) is to order the instructions for maximum ILP.
  - Keep all resources busy every cycle.
  - If necessary, eliminate data dependences and resource hazards to accomplish this.
- The IS problem is NP-complete (and bad in practice).
  - So heuristic methods are necessary.
Instruction Scheduling

• There are many different techniques for IS.
  - Still an open area of research.
• Most optimizing compilers perform good local IS, and only simple global IS.
• The biggest opportunities are in scheduling the code for loops.

Should the Compiler Do IS?

• Many modern machines perform dynamic reordering of instructions.
  - Also called “out-of-order execution” (OOOE).
  - Not yet clear whether this is a good idea.
  - Pro:
    • OOOE can use additional registers and register renaming to eliminate data dependences that no amount of static IS can accomplish.
    • No need to recompile programs when hardware changes.
  - Con:
    • OOOE means more complex hardware (and thus longer cycle times and more wattage).
    • And can’t be optimal since IS is NP-complete.

What we will cover

• Scheduling basic blocks
  - List scheduling
  - Long-latency operations
  - Delay slots
• Software Pipelining
• Scheduling for clusters architectures (next week)

Instruction Scheduling

• In the von Neumann model of execution an instruction starts only after its predecessor completes.
  
  \[ \text{time} \]
  \[
  \begin{array}{c|c}
  \text{instr 1} & \text{instr 2} \\
  \end{array}
  \]

• This is not a very efficient model of execution.
  - von Neumann bottleneck or the memory wall.
Instruction Pipelines

- Almost all processors today use instructions pipelines to allow overlap of instructions (Pentium 4 has a 20 stage pipeline!!!).
- The execution of an instruction is divided into stages; each stage is performed by a separate part of the processor.

- Each of these stages completes its operation in one cycle (shorter than in the von Neumann model).
- An instruction still takes the same time to execute.

F: Fetch instruction from cache or memory.
D: Decode instruction.
E: Execute. ALU operation or address calculation.
M: Memory access.
W: Write back result into register.

Instruction Pipelines

- However, we overlap these stages in time to complete an instruction every cycle.

Filling the pipeline

Draining the pipeline

Steady state

Pipeline Hazards

- Structural Hazards
  - two instructions need the same resource at the same time
  - memory or functional units in a superscalar.
- Data Hazards
  - an instruction needs the results of a previous instruction
    - $r_1 = r_2 + r_3$
    - $r_4 = r_1 + r_1$
    - solved by forwarding and/or stalling
    - cache miss?
- Control Hazards
  - jump & branch address not known until later in pipeline
  - solved by delay slot and/or prediction

Jump/Branch Delay Slot(s)

- Control hazards, i.e. jump/branch instructions.

unconditional jump address available only after Decode.
conditional branch address available only after Execute.

jump/branch

- solved by delay slot and/or prediction
Jump/Branch Delay Slot(s)

- One option is to stall the pipeline (hardware solution).

  jump FDEMW
  instr 2 FDEMW

- Another option is to insert a no-op instruction (software).

  jump FDEMW
  nop FDEMW
  instr 2 FDEMW

- Both degrade performance!

Jump/Branch Delay Slots

- In other words, the instruction(s) in the delay slots of the jump/branch instruction always get(s) executed when the branch is executed (regardless of the branch result).
- Fetching from the branch target begins only after these instructions complete.

  bgt r3, L1
  :
  :
  L1:

Branch Prediction

- Current processors will speculatively execute at conditional branches
  - if a branch direction is correctly guessed, great!
  - if not, the pipeline is flushed before instructions commit (WB).
- Why not just let compiler schedule?
  - The average number of instructions per basic block in typical C code is about 5 instructions.
  - branches are not statically predictable
  - What happens if you have a 20 stage pipeline?
Data Hazards

\[ r1 = r2 + r3 \]
\[ r4 = r1 + r1 \]

Defining Dependencies

- Flow Dependence: \( W \rightarrow R \) \( \delta^f \) \( \{ \text{true} \} \)
- Anti-Dependence: \( R \rightarrow W \) \( \delta^a \) \( \{ \text{false} \} \)
- Output Dependence: \( W \rightarrow W \) \( \delta^o \)
- Input Dependence: \( R \rightarrow R \) \( \delta^i \)

Example Dependencies

\[
\begin{align*}
S1) & \ a=0; \\
S2) & \ b=a; \\
S3) & \ c=a+d+e; \\
S4) & \ d=b; \\
S5) & \ b=5+e;
\end{align*}
\]

- \( S1 \delta^f S2 \) due to a
- \( S1 \delta^f S3 \) due to a
- \( S2 \delta^f S4 \) due to b
- \( S3 \delta^a S4 \) due to d
- \( S4 \delta^a S5 \) due to b
- \( S2 \delta^o S5 \) due to b
- \( S3 \delta^i S5 \) due to a

Renaming of Variables

- Sometimes constraints are not “real,” in the sense that a simple renaming of variables/registers can eliminate them.
  - Output dependence (WW): A and B write to the same variable.
  - Anti dependence (RW): A reads from a variable to which B writes.
- In such cases, the order of A and B cannot be changed unless variables are renamed.
- Can sometimes be done by the hardware, to a limited extent.
Register Renaming Example

\[
\begin{align*}
\text{r1} &\leftarrow \text{r2} + 1 \\
[\text{fp}+8] &\leftarrow \text{r1} \\
\text{r1} &\leftarrow \text{r3} + 2 \\
[\text{fp}+12] &\leftarrow \text{r1}
\end{align*}
\]

Phase ordering problem

- Can perform register renaming after register allocation
  - Constrained by available registers
  - Constrained by live on entry/exit
- Instead, do scheduling before register allocation

Scheduling a BB

- What do we need to know?
  - Latency of operations
  - \# of registers
- Assume:
  - load \ 5
  - store \ 5
  - mult \ 2
  - others \ 1
- Also assume,
  - operations are non-blocking

\[
\begin{align*}
\text{x} &\leftarrow \text{w} \times \text{y} \times \text{z} \\
\text{r1} &\leftarrow [\text{fp}+\text{w}] \\
\text{r2} &\leftarrow 2 \\
\text{r1} &\leftarrow \text{r1} \times \text{r2} \\
\text{r2} &\leftarrow [\text{fp}+\text{x}] \\
\text{r1} &\leftarrow \text{r1} \times \text{r2} \\
\text{r2} &\leftarrow [\text{fp}+\text{y}] \\
\text{r1} &\leftarrow \text{r1} \times \text{r2} \\
\text{r2} &\leftarrow [\text{fp}+\text{z}] \\
[\text{fp}+\text{w}] &\leftarrow \text{r1}
\end{align*}
\]

We can do better

- Assume:
  - load \ 5
  - store \ 5
  - mult \ 2
  - others \ 1
  - operations are non-blocking

\[
\begin{align*}
\text{x} &\leftarrow \text{w} \times \text{y} \times \text{z} \\
1 &\text{r1} \leftarrow [\text{fp}+\text{w}] \\
2 &\text{r2} \leftarrow [\text{fp}+\text{x}] \\
3 &\text{r3} \leftarrow [\text{fp}+\text{y}] \\
4 &\text{r4} \leftarrow [\text{fp}+\text{z}] \\
5 &\text{r5} \leftarrow 2 \\
6 &\text{r1} \leftarrow \text{r1} \times \text{r5} \\
8 &\text{r1} \leftarrow \text{r1} \times \text{r2} \\
10 &\text{r1} \leftarrow \text{r1} \times \text{r3} \\
12 &\text{r1} \leftarrow \text{r1} \times \text{r4} \\
14 &[\text{fp}+\text{w}] \leftarrow \text{r1}
\end{align*}
\]

We can do even better if we assume what?

19 \text{r1 can be used again}
Defining Better

1. \( r_1 \leftarrow [\text{fp+w}] \)
2. \( r_2 \leftarrow 2 \)
6. \( r_1 \leftarrow r_1 \times r_2 \)
7. \( r_2 \leftarrow [\text{fp+x}] \)
12. \( r_1 \leftarrow r_1 \times r_2 \)
13. \( r_2 \leftarrow [\text{fp+y}] \)
18. \( r_1 \leftarrow r_1 \times r_2 \)
19. \( r_2 \leftarrow [\text{fp+z}] \)
24. \( r_1 \leftarrow r_1 \times r_2 \)
26. \( [\text{fp+w}] \leftarrow r_1 \)
33. \( r_1 \) can be used again

The Scheduler

• Given:
  - Code to schedule
  - Resources available (FU and # of Reg)
  - Latencies of instructions
• Goal:
  - Correct code
  - Better code [fewer cycles, less power, fewer registers, …]
  - Do it quickly

More Abstractly

• Given a graph \( G = (V,E) \) where
  - nodes are operations
    • Each operation has an associated delay and type
  - edges between nodes represent dependencies
  - The number of resources of type \( t \), \( R(t) \)
• A schedule assigns to each node a cycle number:
  - \( S(n) \geq 0 \)
  - If \((n,m) \in G\), \( S(m) \geq S(n) + \text{delay(n)} \)
  - \(|\{n \mid S(n) = x \text{ and type(n) = t}\}| \leq R(t)\)
• Goal is shortest length schedule, where length
  - \( L(S) = \max \text{ over } n, S(n) + \text{delay(n)} \)

List Scheduling

• Keep a list of available instructions, I.e.,
  - If we are at cycle \( k \), then all predecessors, \( p \), in graph have all been scheduled so that \( S(p) + \text{delay(p)} \leq k \)
• Pick some instruction, \( n \), from queue such that there are resources for type(\( n \))
• Update available instructions and continue
• It is all in how we pick instructions
Lots of Heuristics

- forward or backward
- choose instructions on critical path
- ASAP or ALAP
- Balanced paths
- depth in schedule graph

DLS (1995)

- Aim: avoid pipeline hazards in load/store unit
  - load followed by use of target reg
  - store followed by load
- Simplifies in two ways
  - 1 cycle latency for load/store
  - includes all dependencies (WaW included)

The algorithm

- Construct Scheduling dag
- Make srcs of dag candidates
- Pick a candidate
  - Choose an instruction with an interlock
  - Choose an instruction with a large number of successors
  - Choose with longest path to root
- Add newly available instruction to candidate list

1) ld r1 ← [a]
2) ld r2 ← [b]
3) add r1 ← r1 + r2
4) ld r2 ← [c]
5) ld r3 ← [d]
6) mul r4 ← r2 * r3
7) add r1 ← r1 + r4
8) add r2 ← r2 + r3
9) mul r2 ← r2 * r3
10) add r1 ← r1 + r2
11) st [a] ← r1
Trace Scheduling

- Basic blocks typically contain a small number of instructions.
- With many FUs, we may not be able to keep all the units busy with just the instructions of a BB.
- Trace scheduling allows block scheduling across BBs.
- The basic idea is to dynamically determine which blocks are executed more frequently. The set of such BBs is called a trace.

The trace is then scheduled as a single BB.
- Blocks that are not part of the trace must be modified to restore program semantics if/when execution goes off-trace.

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

Assume all have latency of 2

\[
\begin{align*}
A & \leftarrow \text{ld}[d] \\
B & \leftarrow a \ast a \\
C & \leftarrow \text{st}[d], b \\
D & \leftarrow d + 4
\end{align*}
\]

Can we decrease the latency?

- Lets unroll

\[
\begin{align*}
A & \leftarrow \text{ld}[d] \\
B & \leftarrow a \ast a \\
C & \leftarrow \text{st}[d], b \\
D & \leftarrow d + 4 \\
A1 & \leftarrow \text{ld}[d1] \\
B1 & \leftarrow a1 \ast a1 \\
C1 & \leftarrow \text{st}[d1], b1 \\
D1 & \leftarrow d1 + 4
\end{align*}
\]

Rename variables

\[
\begin{align*}
A & \leftarrow \text{ld}[d] \\
B & \leftarrow a \ast a \\
C & \leftarrow \text{st}[d], b \\
D & \leftarrow d + 4 \\
A1 & \leftarrow \text{ld}[d1] \\
B1 & \leftarrow a1 \ast a1 \\
C1 & \leftarrow \text{st}[d1], b1 \\
D1 & \leftarrow d1 + 4
\end{align*}
\]

Schedule

\[
\begin{align*}
A & \leftarrow \text{ld}[d] \\
B & \leftarrow a \ast a \\
C & \leftarrow \text{st}[d], b \\
D & \leftarrow d + 4 \\
A1 & \leftarrow \text{ld}[d1] \\
B1 & \leftarrow a1 \ast a1 \\
C1 & \leftarrow \text{st}[d1], b1 \\
D1 & \leftarrow d1 + 4
\end{align*}
\]
Unroll Some More

A: \(a \leftarrow \text{ld}[d]\)
B: \(b \leftarrow a \times a\)
C: \(\text{st}[d], b\)
D: \(d_1 \leftarrow d + 4\)
A1: \(a_1 \leftarrow \text{ld}[d_1]\)
B1: \(b_1 \leftarrow a_1 \times a_1\)
C1: \(\text{st}[d_1], b_1\)
D1: \(d_2 \leftarrow d_1 + 4\)
A2: \(a_2 \leftarrow \text{ld}[d_2]\)
B2: \(b_2 \leftarrow a_2 \times a_2\)
C2: \(\text{st}[d_2], b_2\)
D2: \(d \leftarrow d_2 + 4\)

One More Time

Can Rearrange
Rearrange

A: a ← ld [d]
B: b ← a * a
C: c ← st [d], b
D: d ← d + 4
A1: a1 ← ld [d1]
B1: b1 ← a1 * a1
C1: c1 ← st [d1], b1
D1: d1 ← d1 + 4
A2: a2 ← ld [d2]
B2: b2 ← a2 * a2
C2: c2 ← st [d2], b2
D2: d2 ← d2 + 4

SP Loop

A: a ← ld [d]
B: b ← a * a
C: c ← st [d], b
D: d ← d + 4
A1: a1 ← ld [d1]
B1: b1 ← a1 * a1
C1: c1 ← st [d1], b1
D1: d1 ← d1 + 4
A2: a2 ← ld [d2]
B2: b2 ← a2 * a2
C2: c2 ← st [d2], b2
D2: d2 ← d2 + 4

Goal of SP

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

Assume operating on an infinite wide machine

Example

Assume operating on an infinite wide machine

Dealing with exit conditions

for (i=0; i<N; i++)
{
  A_i
  if (i >= N) goto done
  A_0
  B_0
  if (i+1 == N) goto last
  i=1
  A_1
  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
- 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]
```

**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]
```

**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]
```

**Step 3**

Build a tableau of iteration number vs cycle time.

```
<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>acf</td>
<td>jf</td>
<td>fj</td>
<td>fj</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>bd</td>
<td>egh</td>
<td>a</td>
<td>cb</td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>dg</td>
<td>a</td>
<td>cb</td>
<td>eh</td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>gh</td>
<td>c</td>
<td>b</td>
<td>a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>e</td>
<td>dh</td>
<td>a</td>
<td>g</td>
<td>c</td>
<td></td>
<td></td>
</tr>
<tr>
<td>f</td>
<td>eh</td>
<td>b</td>
<td>a</td>
<td>d</td>
<td></td>
<td></td>
</tr>
<tr>
<td>g</td>
<td>eh</td>
<td>b</td>
<td>a</td>
<td>d</td>
<td></td>
<td></td>
</tr>
<tr>
<td>h</td>
<td>eh</td>
<td>b</td>
<td>a</td>
<td>d</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Aiken/Nicolau Scheduling

Step 4

Find repeating patterns of instructions.

```
1  iteration
1  2  3  4  5  6
1  acfj  fj  fj  fj  fj  fj
2  bd
3  egh  a
4  cb
5  dg  a
6  eh  b
cycle
7  cg  a
d
8  eh  g  a
9  c  b
d  g  a
10  eh  b
t  cycle
11  c  bd  g  a
eh  b
c  gdeh
12  iteration
1  2  3  4  5  6
1  acfj  fj  fj  fj  fj  fj
2  bd
3  egh  a
4  cb
5  dg  a
6  eh  b
cycle
7  cg  a
d
8  eh  g  a
9  c  b
d  g  a
10  eh  b
t  cycle
11  c  bd  g  a
eh  b
c  gdeh
```

Aiken/Nicolau Scheduling

Step 5

"Coalesce" the slopes.

```
1  iteration
1  2  3  4  5  6
1  acfj  fj  fj  fj  fj  fj
2  bd
3  egh  a
4  cb
5  dg  a
6  eh  b
cycle
7  cg  a
d
8  eh  g  a
9  c  b
d  g  a
10  eh  b
t  cycle
11  c  bd  g  a
eh  b
c  gdeh
```

Aiken/Nicolau Scheduling

Step 6

Find the loop body and "reroll" the loop.

```
1  iteration
1  2  3  4  5  6
1  acfj  fj  fj  fj  fj  fj
2  bd
3  egh  a
4  cb
5  dg  a
6  eh  b
cycle
7  cg  a
d
8  eh  g  a
9  c  b
d  g  a
10  eh  b
t  cycle
11  c  bd  g  a
eh  b
c  gdeh
```

Aiken/Nicolau Scheduling

Step 4

Find repeating patterns of instructions.

```
1  iteration
1  2  3  4  5  6
1  acfj  fj  fj  fj  fj  fj
2  bd
3  egh  a
4  cb
5  dg  a
6  eh  b
cycle
7  cg  a
d
8  eh  g  a
9  c  b
d  g  a
10  eh  b
t  cycle
11  c  bd  g  a
eh  b
c  gdeh
```

Aiken/Nicolau Scheduling

Step 5

"Coalesce" the slopes.

```
1  iteration
1  2  3  4  5  6
1  acfj  fj  fj  fj  fj  fj
2  bd
3  egh  a
4  cb
5  dg  a
6  eh  b
cycle
7  cg  a
d
8  eh  g  a
9  c  b
d  g  a
10  eh  b
t  cycle
11  c  bd  g  a
eh  b
c  gdeh
```

Aiken/Nicolau Scheduling

Step 6

Find the loop body and "reroll" the loop.

```
1  iteration
1  2  3  4  5  6
1  acfj  fj  fj  fj  fj  fj
2  bd
3  egh  a
4  cb
5  dg  a
6  eh  b
cycle
7  cg  a
d
8  eh  g  a
9  c  b
d  g  a
10  eh  b
t  cycle
11  c  bd  g  a
eh  b
c  gdeh
```
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>a0</td>
<td>acfj</td>
<td>bd</td>
<td>fuj</td>
<td>egh</td>
<td>a</td>
<td>cb</td>
</tr>
<tr>
<td>d0</td>
<td>c</td>
<td>f0</td>
<td>a</td>
<td>fj</td>
<td>d</td>
<td>a</td>
</tr>
<tr>
<td>6</td>
<td>a</td>
<td>fuj</td>
<td>d</td>
<td>a</td>
<td>c</td>
<td>f0</td>
</tr>
<tr>
<td>7</td>
<td>a</td>
<td>fj</td>
<td>d</td>
<td>a</td>
<td>c</td>
<td>f0</td>
</tr>
<tr>
<td>cycle</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td>e</td>
<td>d</td>
<td>b</td>
<td>e</td>
<td>g</td>
<td>f</td>
<td>j</td>
</tr>
<tr>
<td>d</td>
<td>b</td>
<td>e</td>
<td>g</td>
<td>f</td>
<td>j</td>
<td>c</td>
</tr>
</tbody>
</table>

Prologue/entry code

Loop body

Epilogue/exit code

Step 7

Generate code.

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

\[
\begin{align*}
1 & : = j_0 \oplus b_0 & c_1 & : = e_0 \oplus j_0 & f_1 & : = U[1] & j_1 & : = X[1] \\
b_1 & : = a_1 \oplus f_0 & d_1 & : = f_0 \oplus c_1 & f_2 & : = U[2] & j_2 & : = X[2] \\
e_1 & : = b_1 \oplus d_1 & V[1] & : = b_1 & W[1] & : = d_1 & a_2 & : = j_1 \oplus b_1 \\
c_2 & : = e_1 \oplus j_1 & b_2 & : = a_2 \oplus f_1 & f_3 & : = U[3] & j_3 & : = X[3] \\
d_2 & : = f_1 \oplus c_2 & V[2] & : = b_2 & a_3 & : = j_2 \oplus b_2 \\
c_3 & : = e_2 \oplus j_2 & V[3] & : = b_3 & a_4 & : = j_3 \oplus b_3 & i & = 3 \\
L: \\
d_i & : = f_i-1 \oplus c_i & b_i & : = a_i \oplus f_i \\
e_i & : = b_i \oplus d_i & W[i] & : = d_i & V[i+1] & : = b_{i+1} & f_{i+2} & : = U[i+2] & j_{i+2} & : = X[i+2] \\
c_{i+1} & : = e_i \oplus j_i & a_{i+2} & : = j_{i+1} \oplus b_{i+1} & i & = i+1 & \text{if } i < N-2 \text{ goto } L \\
d_{i+1} & : = f_{i+2} \oplus c_{i+1} & b_{i+1} & : = a_{i+1} \oplus f_{i+1} \\
e_{i+1} & : = b_{i+1} \oplus d_{i+1} & W[N-1] & : = d_{i+1} & V[N] & : = b_{i+1} \\
c_{i+1} & : = e_i \oplus j_i & a_{i+2} & : = j_{i+1} \oplus b_{i+1} & i & = i+1 & \text{if } i < N-2 \text{ goto } L \\
d_{N-1} & : = f_{N-2} \oplus c_{N-1} & b_N & : = a_N \oplus f_{N-1} \\
e_N & : = b_{N} \oplus d_N & W[N-1] & : = d_N & V[N] & : = b_N \\
c_N & : = e_N \oplus j_N \\
e_N & : = b_N \oplus d_N & W[N] & : = d_N \end{align*}
\]

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
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

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 \( r(i) \)

Precedence Constraints

- Data Dependence + Latency of the functional unit being used
- The precedence constraint between two nodes, u and v, is the minimal delay between starting u and v in the schedule.

Software Pipelining Goal

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

• Combine the resource constraints of instructions at steps $i, i+s, i+2s, i+3s$, etc.

Precedence Constraints

• Constraint becomes a tuple: $<p,d>$
  - $p$ is the minimum iteration delay (or the loop carried dependence distance)
  - $d$ is the delay
• For an edge, $u \rightarrow v$, we must have $\sigma(v) - \sigma(u) \geq d(u,v) - s \cdot p(u,v)$
  - $p \geq 0$
  - If data dependence is loop
    - independent $p=0$
    - loop-carried $p>0$

Iterative Approach

• 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

Lower Bounds

• Resource Constraints: $S_R$
  maximum over all resources of # of uses divided by # available
  What is lower bound. Is it tight?

• Precedence Constraints: $S_E$
  max over all cycles: $d(c)/p(c)$
**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
  - $\sigma[n] = \text{the time at which } n \text{ is scheduled or NONE}$
- Prioritize instructions by some heuristic
  - critical path
  - resource critical

**Scheduling algorithm**

- pick an instruction, $n$
- Calculate earliest time due to dependence constraints
  For all $x = \text{pred}(n)$,
    \[
    \text{earliest} = \max(\text{earliest}, \sigma(x) + d(x,n) - sp(x,n))
    \]
- try and schedule $n$ from earliest to earliest+$s$-1 s.t. resource constraints are obeyed.
- If we fail, then this schedule is faulty
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.
    \( \sigma(x) \geq \sigma(n) + d(n,x) - sp(n,x) \), otherwise they are removed.
  - All scheduled instructions (except \( n \)) that have data dependence conflicts are removed.
- repeat this some number of times until either
  - succeed, then register allocate
  - fail, then increase \( s \)

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: ?

```
instr | \( \sigma \)
---|---
a | <1,1>
b | <0,1>
c | <1,1>
d | <0,1>
e | <0,1>
f | <0,1>
g | <0,1>
h | <0,1>
j | <0,1>
```

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

```
instr | \( \sigma \)
---|---
a | <1,1>
b | <0,1>
c | <1,1>
d | <0,1>
e | <0,1>
f | <0,1>
g | <0,1>
h | <1,1>
j | <1,1>
```

s=5
for $i:=1$ to $N$ do
    $a := j \oplus b$
    $b := a \oplus f$
    $c := e \oplus j$
    $d := f \oplus c$
    $e := b \oplus d$
    $f := U[i]$
    $g := V[i] := b$
    $h := W[i] := d$
    $j := X[i]$

Priorities: $a,b,f,j,g,h$

\[
\begin{array}{|c|c|}
\hline
\text{instr} & \sigma \\
\hline
a & 3 \\
b & 4 \\
c & 0 \\
d & 1 \\
e & 2 \\
f & \\
g & \\
h & \\
i & \\
\hline
\end{array}
\]

Priorities: $b,f,j,g,h$

$b$ causes $b \rightarrow e$ edge violation
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,j,g,h

e causes e->c edge violation

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: f,j,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: j,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: g,h
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

<table>
<thead>
<tr>
<th>instr</th>
<th>( \sigma )</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>3</td>
</tr>
<tr>
<td>b</td>
<td>4</td>
</tr>
<tr>
<td>c</td>
<td>5</td>
</tr>
<tr>
<td>d</td>
<td>6</td>
</tr>
<tr>
<td>e</td>
<td>7</td>
</tr>
<tr>
<td>f</td>
<td>0</td>
</tr>
<tr>
<td>g</td>
<td>7</td>
</tr>
<tr>
<td>h</td>
<td>8</td>
</tr>
<tr>
<td>j</td>
<td>1</td>
</tr>
</tbody>
</table>

\[
f_0 = U[0]; \\
j_0 = X[0]; \\
\text{FOR } i = 0 \text{ to } N \\
f_1 := U[i+1] \\
j_1 := X[i+1] \\
\text{nop} \\
a := j_0 ? b \\
b := a ? f_0 \\
c := e ? j_0 \\
d := f_0 ? c \\
e := b ? d \\
g: V[i] := b \\
h: W[i] := d \\
f_0 = f_1 \\
j_0 = j_1
\]

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