Lecture: Static ILP

• Topics: loop unrolling, VLIW, software pipelines, predication
Scheduled and Unrolled Loop

<table>
<thead>
<tr>
<th>Loop:</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>L.D F0, 0(R1)</td>
</tr>
<tr>
<td>2.</td>
<td>L.D F6, -8(R1)</td>
</tr>
<tr>
<td>3.</td>
<td>L.D F10, -16(R1)</td>
</tr>
<tr>
<td>4.</td>
<td>L.D F14, -24(R1)</td>
</tr>
<tr>
<td>5.</td>
<td>ADD.D F4, F0, F2</td>
</tr>
<tr>
<td>6.</td>
<td>ADD.D F8, F6, F2</td>
</tr>
<tr>
<td>7.</td>
<td>ADD.D F12, F10, F2</td>
</tr>
<tr>
<td>8.</td>
<td>ADD.D F16, F14, F2</td>
</tr>
<tr>
<td>9.</td>
<td>S.D F4, 0(R1)</td>
</tr>
<tr>
<td>10.</td>
<td>S.D F8, -8(R1)</td>
</tr>
<tr>
<td>11.</td>
<td>DADDUI R1, R1, # -32</td>
</tr>
<tr>
<td>12.</td>
<td>S.D F12, 16(R1)</td>
</tr>
<tr>
<td>13.</td>
<td>BNE R1,R2, Loop</td>
</tr>
<tr>
<td>14.</td>
<td>S.D F16, 8(R1)</td>
</tr>
</tbody>
</table>

- Execution time: 14 cycles or 3.5 cycles per original iteration

LD -> any : 1 stall
FPALU -> any: 3 stalls
FPALU -> ST : 2 stalls
IntALU -> BR : 1 stall
Loop Unrolling

- Increases program size

- Requires more registers

- To unroll an n-iteration loop by degree k, we will need \((n/k)\) iterations of the larger loop, followed by \((n \mod k)\) iterations of the original loop
Automating Loop Unrolling

- Determine the dependences across iterations: in the example, we knew that loads and stores in different iterations did not conflict and could be re-ordered

- Determine if unrolling will help – possible only if iterations are independent

- Determine address offsets for different loads/stores

- Dependency analysis to schedule code without introducing hazards; eliminate name dependences by using additional registers
Problem 2

for (i=1000; i>0; i--)
x[i] = y[i] * s;

Source code

<table>
<thead>
<tr>
<th>Loop:</th>
<th>Instruction</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>L.D</td>
<td>F0, 0(R1)</td>
<td>F0 = array element</td>
</tr>
<tr>
<td>MUL.D</td>
<td>F4, F0, F2</td>
<td>multiply scalar</td>
</tr>
<tr>
<td>S.D</td>
<td>F4, 0(R2)</td>
<td>store result</td>
</tr>
<tr>
<td>DADDUI</td>
<td>R1, R1,#-8</td>
<td>decrement address pointer</td>
</tr>
<tr>
<td>DADDUI</td>
<td>R2, R2,#-8</td>
<td>decrement address pointer</td>
</tr>
<tr>
<td>BNE</td>
<td>R1, R3, Loop</td>
<td>branch if R1 != R3</td>
</tr>
<tr>
<td>NOP</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Assembly code

• How many unrolls does it take to avoid stall cycles?
Problem 2

for (i=1000; i>0; i--)
x[i] = y[i] * s;

Loop: L.D F0, 0(R1) ; F0 = array element
      MUL.D F4, F0, F2 ; multiply scalar
      S.D F4, 0(R2) ; store result
      DADDUI R1, R1,# -8 ; decrement address pointer
      DADDUI R2, R2,#-8 ; decrement address pointer
      BNE R1, R3, Loop ; branch if R1 != R3
      NOP

• How many unrolls does it take to avoid stall cycles?

Degree 2: LD LD MUL MUL DA DA 1s SD BNE SD
Degree 3: LD LD LD MUL MUL MUL DA DA SD SD BNE SD
– 12 cyc/3 iterations
### Superscalar Pipelines

<table>
<thead>
<tr>
<th>Integer pipeline</th>
<th>FP pipeline</th>
</tr>
</thead>
<tbody>
<tr>
<td>Handles L.D, S.D, ADDUI, BNE</td>
<td>Handles ADD.D</td>
</tr>
</tbody>
</table>

- What is the schedule with an unroll degree of 5?
Superscalar Pipelines

<table>
<thead>
<tr>
<th>Loop:</th>
<th>Integer pipeline</th>
<th>FP pipeline</th>
</tr>
</thead>
<tbody>
<tr>
<td>L.D</td>
<td>F0,0(R1)</td>
<td></td>
</tr>
<tr>
<td>L.D</td>
<td>F6,-8(R1)</td>
<td>ADD.D</td>
</tr>
<tr>
<td>L.D</td>
<td>F10,-16(R1)</td>
<td>F4,F0,F2</td>
</tr>
<tr>
<td>L.D</td>
<td>F14,-24(R1)</td>
<td>ADD.D</td>
</tr>
<tr>
<td>L.D</td>
<td>F18,-32(R1)</td>
<td>F8,F6,F2</td>
</tr>
<tr>
<td>S.D</td>
<td>F4,0(R1)</td>
<td>ADD.D</td>
</tr>
<tr>
<td>S.D</td>
<td>F8,-8(R1)</td>
<td>F12,F10,F2</td>
</tr>
<tr>
<td>S.D</td>
<td>F12,-16(R1)</td>
<td>ADD.D</td>
</tr>
<tr>
<td>DADDUI</td>
<td>R1,R1,# -40</td>
<td>F16,F14,F2</td>
</tr>
<tr>
<td>S.D</td>
<td>F16,16(R1)</td>
<td>ADD.D</td>
</tr>
<tr>
<td>BNE</td>
<td>R1,R2,Loop</td>
<td>F20,F18,F2</td>
</tr>
<tr>
<td>S.D</td>
<td>F20,8(R1)</td>
<td></td>
</tr>
</tbody>
</table>

- Need unroll by degree 5 to eliminate stalls (fewer if we move DADDUI up)
- The compiler may specify instructions that can be issued as one packet
- The compiler may specify a fixed number of instructions in each packet: Very Large Instruction Word (VLIW)
Problem 3

for (i=1000; i>0; i--)
    x[i] = y[i] * s;

Source code

Loop:     L.D         F0, 0(R1)          ; F0 = array element
          MUL.D    F4, F0, F2        ; multiply scalar
          S.D         F4, 0(R2)          ; store result
          DADDUI  R1, R1,# -8      ; decrement address pointer
          DADDUI  R2, R2,#-8       ; decrement address pointer
          BNE        R1, R3, Loop    ; branch if R1 != R3
          NOP

Assembly code

• How many unrolls does it take to avoid stalls in the superscalar pipeline?
Problem 3

for (i=1000; i>0; i--)
    x[i] = y[i] * s;

Loop:  L.D F0, 0(R1) ; F0 = array element
       MUL.D F4, F0, F2 ; multiply scalar
       S.D F4, 0(R2) ; store result
       DADDUI R1, R1,# -8 ; decrement address pointer
       DADDUI R2, R2,# -8 ; decrement address pointer
       BNE R1, R3, Loop ; branch if R1 != R3
       NOP

• How many unrolls does it take to avoid stalls in the superscalar pipeline?

    LD
    LD
    LD  MUL
    LD  MUL
    LD  MUL
    7 unrolls. Could also make do with 5 if we
    moved up the DADDUIs.
Loop: L.D  F0, 0(R1)
ADD.D  F4, F0, F2
S.D    F4, 0(R1)
DADDUI R1, R1,# -8
BNE    R1, R2, Loop
Software Pipelining

• Advantages: achieves nearly the same effect as loop unrolling, but without the code expansion – an unrolled loop may have inefficiencies at the start and end of each iteration, while a sw-pipelined loop is almost always in steady state – a sw-pipelined loop can also be unrolled to reduce loop overhead

• Disadvantages: does not reduce loop overhead, may require more registers

Loop:

L.D  F0, 0(R1)
ADD.D  F4, F0, F2
S.D  F4, 0(R1)
DADDUI  R1, R1,# -8
BNE  R1, R2, Loop

Loop:

S.D  F4, 16(R1)
ADD.D  F4, F0, F2
L.D  F0, 0(R1)
DADDUI  R1, R1,# -8
BNE  R1, R2, Loop
Problem 4

Source code

```c
for (i=1000; i>0; i--)
    x[i] = y[i] * s;
```

Assembly code

```
Loop:     L.D         F0, 0(R1)          ; F0 = array element
          MUL.D    F4, F0, F2        ; multiply scalar
          S.D         F4, 0(R2)          ; store result
          DADDUI  R1, R1,# -8      ; decrement address pointer
          DADDUI  R2, R2,#-8       ; decrement address pointer
          BNE        R1, R3, Loop    ; branch if R1 != R3
          NOP
```

- LD -> any : 1 stall
- FPMUL -> any: 5 stalls
- FPMUL -> ST : 4 stalls
- IntALU -> BR : 1 stall

Show the SW pipelined version of the code and does it cause stalls?
Problem 4

for (i=1000; i>0; i--)
  \( x[i] = y[i] \times s; \)

Source code

Assembly code

Loop:  
  L.D  F0, 0(R1)  ; F0 = array element  
  MUL.D F4, F0, F2  ; multiply scalar  
  S.D  F4, 0(R2)  ; store result  
  DADDUI R1, R1, # -8  ; decrement address pointer  
  DADDUI R2, R2, #-8  ; decrement address pointer  
  BNE  R1, R3, Loop  ; branch if R1 != R3  
  NOP

• Show the SW pipelined version of the code and does it cause stalls?

Loop:  
  S.D  F4, 0(R2)  
  MUL  F4, F0, F2  
  L.D  F0, 0(R1)  
  DADDUI R2, R2, # -8  
  BNE  R1, R3, Loop  
  DADDUI R1, R1, # -8  

There will be no stalls
Predication

- A branch within a loop can be problematic to schedule
- Control dependences are a problem because of the need to re-fetch on a mispredict
- For short loop bodies, control dependences can be converted to data dependences by using predicated/conditional instructions
Predicated or Conditional Instructions

if \((R1 == 0)\)
\[
\begin{align*}
R2 &= R2 + R4 \\
else \\
R6 &= R3 + R5 \\
R4 &= R2 + R3
\end{align*}
\]

\[
\begin{align*}
R7 &= !R1 \\
R8 &= R2 \\
R2 &= R2 + R4 \quad \text{(predicated on R7)} \\
R6 &= R3 + R5 \quad \text{(predicated on R1)} \\
R4 &= R8 + R3 \quad \text{(predicated on R1)}
\end{align*}
\]
Predicated or Conditional Instructions

• The instruction has an additional operand that determines whether the instr completes or gets converted into a no-op

• Example: lwc R1, 0(R2), R3  (load-word-conditional) will load the word at address (R2) into R1 if R3 is non-zero; if R3 is zero, the instruction becomes a no-op

• Replaces a control dependence with a data dependence (branches disappear) ; may need register copies for the condition or for values used by both directions

```plaintext
if (R1 == 0)
    R2 = R2 + R4
else
    R6 = R3 + R5
R4 = R2 + R3
```

```plaintext
R7 = !R1 ;  R8 = R2 ;
R2 = R2 + R4  (predicated on R7)
R6 = R3 + R5  (predicated on R1)
R4 = R8 + R3  (predicated on R1)
```
Problem 1

- Use predication to remove control hazards in this code

```plaintext
if (R1 == 0)
  R2 = R5 + R4
  R3 = R2 + R4
else
  R6 = R3 + R2
```
Problem 1

• Use predication to remove control hazards in this code

```plaintext
if (R1 == 0)
   R2 = R5 + R4
   R3 = R2 + R4
else
   R6 = R3 + R2
   R7 = !R1;
   R6 = R3 + R2 (predicated on R1)
   R2 = R5 + R4 (predicated on R7)
   R3 = R2 + R4 (predicated on R7)
```

```plaintext
R7 = !R1;
R6 = R3 + R2 (predicated on R1)
R2 = R5 + R4 (predicated on R7)
R3 = R2 + R4 (predicated on R7)
```
Complications

- Each instruction has one more input operand – more register ports/bypassing

- If the branch condition is not known, the instruction stalls (remember, these are in-order processors)

- Some implementations allow the instruction to continue without the branch condition and squash/complete later in the pipeline – wasted work

- Increases register pressure, activity on functional units

- Does not help if the br-condition takes a while to evaluate
Support for Speculation

- When re-ordering instructions, we need hardware support
  - to ensure that an exception is raised at the correct point
  - to ensure that we do not violate memory dependences
Detecting Exceptions

• Some exceptions require that the program be terminated (memory protection violation), while other exceptions require execution to resume (page faults)

• For a speculative instruction, in the latter case, servicing the exception only implies potential performance loss

• In the former case, you want to defer servicing the exception until you are sure the instruction is not speculative

• Note that a speculative instruction needs a special opcode to indicate that it is speculative