Lecture: Static ILP

• Topics: predication, speculation (Sections C.5, 3.2)
Scheduled and Unrolled Loop

<table>
<thead>
<tr>
<th>Loop:</th>
<th>L.D</th>
<th>F0, 0(R1)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>L.D</td>
<td>F6, -8(R1)</td>
</tr>
<tr>
<td></td>
<td>L.D</td>
<td>F10,-16(R1)</td>
</tr>
<tr>
<td></td>
<td>L.D</td>
<td>F14, -24(R1)</td>
</tr>
<tr>
<td></td>
<td>ADD.D</td>
<td>F4, F0, F2</td>
</tr>
<tr>
<td></td>
<td>ADD.D</td>
<td>F8, F6, F2</td>
</tr>
<tr>
<td></td>
<td>ADD.D</td>
<td>F12, F10, F2</td>
</tr>
<tr>
<td></td>
<td>ADD.D</td>
<td>F16, F14, F2</td>
</tr>
<tr>
<td></td>
<td>S.D</td>
<td>F4, 0(R1)</td>
</tr>
<tr>
<td></td>
<td>S.D</td>
<td>F8, -8(R1)</td>
</tr>
<tr>
<td></td>
<td>DADDUI</td>
<td>R1, R1, # -32</td>
</tr>
<tr>
<td></td>
<td>S.D</td>
<td>F12, 16(R1)</td>
</tr>
<tr>
<td></td>
<td>BNE</td>
<td>R1,R2, Loop</td>
</tr>
<tr>
<td></td>
<td>S.D</td>
<td>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
Problem 2

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 stall cycles?

LD -> any : 1 stall
FPMUL -> any: 5 stalls
FPMUL -> ST : 4 stalls
IntALU -> BR : 1 stall
Problem 2

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

- Source code

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

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

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

Degree 2: LD LD MUL MUL DA DA 1s SD SD BNE SD
Degree 3: LD LD LD MUL 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

- 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
LD  MUL
LD  MUL
LD  MUL
SD  MUL

7 unrolls. Could also make do with 5 if we moved up the DADDUIs.
Software Pipeline?!
Software Pipeline
Software Pipelining

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

• 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
Problem 4

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

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

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

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

• 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       ; branch if R1 != R3
          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)
  R2 = R2 + R4
else
  R6 = R3 + R5
  R4 = R2 + R3

R7 = !R1
R8 = R2

R2 = R2 + R4  (predicated on R7)
R6 = R3 + R5  (predicated on R1)
R4 = R8 + R3  (predicated on R1)
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.

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

```
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

```java
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
   R2 = R5 + R4
   R3 = R2 + R4

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

• In general, when we re-order instructions, register renaming can ensure we do not violate register data dependences.

• However, 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.
Program-Terminate Exceptions

• When a speculative instruction experiences an exception, instead of servicing it, it writes a special NotAThing value (NAT) in the destination register.

• If a non-speculative instruction reads a NAT, it flags the exception and the program terminates (it may not be desirable that the error is caused by an array access, but the segfault happens two procedures later).

• Alternatively, an instruction (the *sentinel*) in the speculative instruction’s original location checks the register value and initiates recovery.
Memory Dependence Detection

• If a load is moved before a preceding store, we must ensure that the store writes to a non-conflicting address, else, the load has to re-execute

• When the speculative load issues, it stores its address in a table (Advanced Load Address Table in the IA-64)

• If a store finds its address in the ALAT, it indicates that a violation occurred for that address

• A special instruction (the sentinel) in the load’s original location checks to see if the address had a violation and re-executes the load if necessary
Problem 2

• For the example code snippet below, show the code after the load is hoisted:

  Instr-A
  Instr-B
  ST R2 → [R3]
  Instr-C
  BEZ R7, foo
  Instr-D
  LD R8 ← [R4]
  Instr-E
Problem 2

For the example code snippet below, show the code after the load is hoisted:

```
Instr-A
Instr-B
ST R2 → [R3]
Instr-C
BEZ R7, foo
Instr-D
LD R8 ← [R4]
Instr-E
```

```
LD.S R8 ← [R4]
Instr-A
Instr-B
ST R2 → [R3]
Instr-C
BEZ R7, foo
Instr-D
LD.C R8, rec-code
Instr-E
```

rec-code: LD R8 ← [R4]
Title

• Bullet