Chapter 3: Instruction Level Parallelism (ILP) and its exploitation

- Pipeline CPI = Ideal pipeline CPI + stalls due to hazards
  - invisible to programmer (unlike process level parallelism)
- ILP: overlap execution of unrelated instructions
  - invisible to programmer (unlike process level parallelism)
- Parallelism within a basic block is limited (a branch every 3-6 instructions)
  - Hence, must explore ILP across basic blocks
- May explore loop level parallelism (fake control dependences) through
  - Loop unrolling (static, compiler based solution)
  - Using vector processors or SIMD architectures
  - Dynamic loop unrolling
- The main challenge is overcoming data and control dependencies

Types of dependences

- **True data dependences**: may cause RAW hazards.
  - Instruction Q uses data produced by instruction P or by an instruction which is data dependent on P.
  - dependences are properties of programs, while hazards are properties of architectures.
  - easy to determine for registers but hard to determine for memory locations since addresses are computed dynamically
    EX: is 100(R1) the same location as 200(R2) or even 100(R1)?

- **Name dependences**: two instructions use the same name but do not exchange data (no data dependency)
  - **Anti-dependence**: Instruction P reads from a register (or memory) followed by instruction Q writing to that register (or memory).
    May cause WAR hazards.
  - **Output dependence**: Instructions P and Q write to the same location.
    May cause WAW hazards.
How can you remove name dependences?

Rename the dependent uses of F0 and F4

Register renaming

How can you remove name dependences?

Rename the dependent uses of F0 and F4
Control dependences

- Determine the order of instructions with respect to branches.

\[
\begin{align*}
\text{if } P_1 \text{ then } S_1 ; \\
\text{if } P_2 \text{ then } S_2 ;
\end{align*}
\]

S1 is control dependent on P1 and S2 is control dependent on P2 (and P1 ??).

- An instruction that is control dependent on P cannot be moved to a place where it is no longer control dependent on P, and visa-versa.

Example 1:

\[
\begin{align*}
\text{add } & R1,R2,R3 \\
\text{beq } & R4, R0, L \\
\text{sub } & R1,R1,R6 \\
L: & \ldots \\
\text{or } & R7,R1,R8
\end{align*}
\]

“or” instruction depends on the execution flow

Example 2:

\[
\begin{align*}
\text{add } & R1,R2,R3 \\
\text{beq } & R12, R0, \text{skip} \\
\text{sub } & R4,R5,R6 \\
\text{add } & R5,R4,R9 \\
\text{skip: } & \ldots \\
\text{or } & R7,R8,R9
\end{align*}
\]

Possible to move “sub” before the branch (if R4 is not used after skip)

Loop carried dependences

For \(i=1,100\)

\[
a[i+1] = a[i] + c[i]
\]

There is a loop carried dependence since the statement in an iteration depends on an earlier iteration.

For \(i=1,100\)

\[
a[i] = a[i] + s
\]

There is no loop carried dependence

- The iterations of a loop can be executed in parallel if there is no \textit{Loop carried dependences}.
The effect of data dependencies (hazards)

- Consider the following loop, which adds a constant to the elements of an array.

```
Loop: fld F0, 0(R1) ; Load an element
      fadd.d F4, F0, F2 ; add a scalar, in F2, to the element
      fsd F4, 0(R1) ; store result
      addi R1, R1, #-8 ; update loop index, R1
      bne R1, R2, Loop ; branch if have not visited all elements
```

- Assume a MIPS pipeline with the following latencies (among others)
  - Latency = 3 cycles if an FP ALU op follows and depends on an FP ALU op.
  - Latency = 2 cycles if an FP store follows and depends on an FP ALU op.
  - Latency = 1 cycle if an FP ALU op follows and depends on an FP load.
  - Latency = 1 cycle if a branch follows and depends on an Integer ALU op.

Pipeline stalls due to data hazards

```
Loop: fld F0, 0(R1) ; Load an element
      stall
      fadd.d F4, F0, F2 ; add a scalar to the array element
      stall
      stall
      fsd F4, 0(R1) ; store result
      addi R1, R1, #-8 ; update loop index, R1
      stall
      bne R1, R2, Loop ; branch if have not visited all elements
```

9 clock cycles per iteration
Basic compiler techniques for exposing ILP (§3.2)

Reorder the statements:

```
Loop:  fld   F0, 0(R1)
       addi  R1, R1, #-8
       fadd.d F4, F0, F2
       stall
       stall
       fsd   F4, 8(R1)
       bne   R1, R2, Loop
```

7 clock cycles per iteration

Note the change from 0 to 8

---

Loop Unrolling (assume no pipelining)

```
Loop:  fld   F0, 0(R1)
       fadd.d F4, F0, F2
       fsd   F4, 0(R1)
       addi  R1, R1, #-8
       fld   F0, 0(R1)
       fadd.d F4, F0, F2
       fsd   F4, 0(R1)
       addi  R1, R1, #-8
       bne   R1, R2, Loop
```

Save 0.5 instruction per iteration

Save 1 instruction per iteration

- Need to worry about boundary cases (strip mining??)
- Can reorder the statements if we use additional registers.
- What limits the number of times we unroll a loop?
- Note that loop iterations were independent

```
for i = 1, 100
x(i) = x(i) + c
```

(10)
Executing the unrolled loop on a pipeline

Loop:
```
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Source1</th>
<th>Source2</th>
</tr>
</thead>
<tbody>
<tr>
<td>fld F0</td>
<td>0(R1)</td>
<td></td>
</tr>
<tr>
<td>fld F6</td>
<td>-8(R1)</td>
<td></td>
</tr>
<tr>
<td>fadd.d F4</td>
<td>F0, F2</td>
<td></td>
</tr>
<tr>
<td>fadd.d F8</td>
<td>F6, F2</td>
<td></td>
</tr>
<tr>
<td>addi R1</td>
<td>R1, R1, #16</td>
<td></td>
</tr>
<tr>
<td>fsd F4</td>
<td>16(R1)</td>
<td></td>
</tr>
<tr>
<td>fsd F8</td>
<td>8(R1)</td>
<td></td>
</tr>
<tr>
<td>bne R1, R2, Loop</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Potential problem if one cycle is required between integer op. and store (see comment below).

4 (or 4.5) clock cycles per iteration

- What can you do if there is no forwarding path from the int/Mem buffer to the ID/int buffer?
  - Stall for one cycle
  - Replace 16(R1) in the first fsd statement by 0(R1).
- What do you do if the latency = 3 cycles when “fsd” follows “fadd.d”? (11)