Instruction Level Parallelism (ILP)

or

Declaration of Independence

What is ILP?

- The characteristic of a program that certain instructions are independent, and can potentially be executed in parallel.
- Any mechanism that creates, identifies, or exploits the independence of instructions, allowing them to be executed in parallel.

- Why do we want/need ILP?
  - In a superscalar architecture?
  - What about a scalar architecture?

Where do we find ILP?

- In basic blocks?
  - 15-20% of (dynamic) instructions are branches in typical code
- Across basic blocks?
  - how?

  for (i=1; i<=1000; i++)
  x[i] = x[i] * s

How do we expose ILP?

- by moving instructions around.
- How??
  - software
  - hardware
Exposing ILP in software

- instruction scheduling (changes ILP within a basic block)
- loop unrolling (allows ILP across iterations by putting instructions from multiple iterations in the same basic block)
- Others (trace scheduling, software pipelining) we’ll talk about later…

A sample loop

```
Loop: LD F0,0(R1) ;F0=array element, R1=X[
MULD F4,F0,F2 ;multiply scalar in F2
SD F4,0(R1) ;store result
ADDI R1,R1,8 ;increment pointer 8B (DW)
SEQ R3, R1, R2 ;R2 = &X[1001]
BNEZ R3,Loop ;branch R3!=zero
NOP ;delayed branch slot
```

Where are the dependencies and stalls?

<table>
<thead>
<tr>
<th>Operation</th>
<th>Latency (stalls)</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP Mult</td>
<td>6 (5)</td>
</tr>
<tr>
<td>LD</td>
<td>2 (1)</td>
</tr>
<tr>
<td>Int ALU</td>
<td>1 (0)</td>
</tr>
</tbody>
</table>

Instruction Scheduling

<table>
<thead>
<tr>
<th>Loop: LD F0,0(R1)</th>
<th>MULD F4,F0,F2</th>
<th>SD 0(R1),F4</th>
<th>ADDI R1,R1,8</th>
<th>MULD F4,F0,F2</th>
<th>SEQ R3, R1, R2</th>
<th>BNEZ R3,Loop</th>
<th>SD -8(R1),F4</th>
</tr>
</thead>
</table>

Loop Unrolling

```
Loop: LD F0,0(R1)
ADDI R1,R1,8
MULD F4,F0,F2
SEQ R3, R1, R2
BNEZ R3,Loop
SD -8(R1),F4
```

CSE 240A Dean Tullsen

CSE 240A Dean Tullsen
Loop Unrolling

Loop:
- LD   F0,0(R1)
- ADDI R1,R1,8
- MULD F4,F0,F2
- SEQ  R3, R1, R2
- BNEZ R3,Loop
- SD   -8(R1),F4

Loop:
- LD   F0,0(R1)
- ADDI R1,R1,8
- MULD F4,F0,F2
- SEQ  R3, R1, R2
- BNEZ R3,Loop
- SD   -8(R1),F4

Register Renaming

Loop:
- LD   F0,0(R1)
- ADDI R1,R1,8
- MULD F4,F0,F2
- SEQ  R3, R1, R2
- BNEZ R3,Loop
- SD   -8(R1),F4

Loop:
- LD   F0,0(R1)
- ADDI R1,R1,8
- MULD F4,F0,F2
- SEQ  R3, R1, R2
- BNEZ R3,Loop
- SD   -8(R1),F4

Loop:
- LD   F0,0(R1)
- ADDI R1,R1,8
- MULD F4,F0,F2
- SEQ  R3, R1, R2
- BNEZ R3,Loop
- SD   -8(R1),F4

Loop:
- LD   F0,0(R1)
- ADDI R1,R1,8
- MULD F4,F0,F2
- SEQ  R3, R1, R2
- BNEZ R3,Loop
- SD   -8(R1),F4

CSE 240A  Dean Tullsen

CSE 240A  Dean Tullsen

CSE 240A  Dean Tullsen

CSE 240A  Dean Tullsen
• Remember: **dependencies** are a property of code, whether or not it is a HW **hazard** depends on the given pipeline.

• Compiler must respect (**True**) Data dependencies (RAW)
  – Instruction i produces a result used by instruction j, or
  – Instruction j is data dependent on instruction k, and instruction k is data dependent on instruction i.
  – Easy to determine for registers (fixed names)
  – Hard for memory:
    • Does 100(R4) = 20(R6)?
    • From different loop iterations, does 20(R6) = 20(R6)?

• Name Dependence Also Harder for Memory Accesses
  – Does 100(R4) = 20(R6)?
  – From different loop iterations, does 20(R6) = 20(R6)?

• As we saw before:
  • Other kinds of dependence also called *false dependence*: two instructions use same *name* but don’t exchange data
  • (WAR dependence)
    – Instruction j writes a register or memory location that instruction i reads from and instruction i is executed first
  • (WAW dependence)
    – Instruction i and instruction j write the same register or memory location; ordering between instructions must be preserved.

• Compilers must also preserve **control dependence**
  • Example
    
    ```c
    if (c1)
    {I1;
     if (c2)
     {I2;
    }
    ```

    I1 is control dependent on c1 and I2 is control dependent on c2 but not on c1.
Compiler Perspectives on
Code Movement

- Two (obvious) constraints on control dependences:
  - An instruction that is control dependent on a branch cannot be moved before the branch so that its execution is no longer controlled by the branch.
  - An instruction that is not control dependent on a branch cannot be moved to after the branch so that its execution is controlled by the branch.

- Control dependencies relaxed to get parallelism; as long as we get same effect if preserve order of exceptions and data flow.

HW Schemes: Instruction Parallelism

- Why in HW at run time?
  - Works when can’t know dependence until run time
    - Variable latency
    - Control dependent data dependence
  - Can schedule differently every time through the code.
  - Compiler simpler
  - Code for one machine runs well on another

- Key idea: Allow instructions behind stall to proceed
  D1VD F0,F2,F4
  ADDD F10,F0,F8
  SUBD F12,F8,F14
  - Enables out-of-order execution => out-of-order completion

First HW ILP Technique:
Out-of-order Issue/Dynamic Scheduling

- Problem -- need to get stalled instructions out of the ID stage, so that subsequent instructions can begin execution.
- Must separate detection of structural hazards from detection of data hazards
- Must split ID operation into two:
  - Issue (decode, check for structural hazards)
  - Read operands (read operands when NO DATA HAZARDS)
- i.e., must be able to issue even when a data hazard exists
- instructions issue in-order, but proceed to EX out-of-order
Dynamic Scheduling by hand

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIVD F0, F2, F4</td>
<td>(10 cycles)</td>
<td></td>
</tr>
<tr>
<td>ADDD F10, F0, F8</td>
<td>(4 cycles)</td>
<td></td>
</tr>
<tr>
<td>SUBD F12, F8, F14</td>
<td>(4 cycles)</td>
<td></td>
</tr>
<tr>
<td>ADDD F20, F2, F3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MULTD F13, F12, F2</td>
<td>(6 cycles)</td>
<td></td>
</tr>
<tr>
<td>ADDD F4, F1, F3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADDD F5, F4, F13</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(assume several FP ADD units)

Key Points

- You can find, create, and exploit Instruction Level Parallelism in SW or HW
- Loop level parallelism is usually easiest to see
- Dependencies exist in a program, and become hazards if HW cannot resolve
- SW dependencies/compiler sophistication determine if compiler can/should unroll loops
- SW code motion is limited by lack of runtime knowledge of dependencies (esp. memory), latencies (esp. memory), and control flow.