Instruction Level Parallelism

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.

**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)

A sample loop

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Result Producing Instruction</th>
<th>Latency (Stalls) in Clock Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD F0.0(R1)</td>
<td>;F0=element, R1=X[]</td>
<td></td>
</tr>
<tr>
<td>MULD F4,F0,F2</td>
<td>;multiply scalar in F2</td>
<td>6</td>
</tr>
<tr>
<td>SD 0(R1),F4</td>
<td>;store result</td>
<td>6</td>
</tr>
<tr>
<td>ADDI R1,R1,8</td>
<td>;increment pointer 8B (DW)</td>
<td>8</td>
</tr>
<tr>
<td>SEQ R3, R1, R2</td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>BNEZ R3,Loop</td>
<td>;branch R3!=zero</td>
<td>8</td>
</tr>
<tr>
<td>NOP</td>
<td>;delayed branch slot</td>
<td>8</td>
</tr>
</tbody>
</table>

Where are the dependencies and stalls?

Instruction Scheduling

- stalls?

Loop Unrolling

- stalls?
Register Renaming

- stalls?

Compiler Perspectives on Code Movement

- 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)?

- Other kinds of dependence also called name dependence: two instructions use same name but don’t exchange data

- Antidependence (WAR dependence)
  - Instruction j writes a register or memory location that instruction i reads from and instruction i is executed first

- Output dependence (WAW dependence)
  - Instruction i and instruction j write the same register or memory location; ordering between instructions must be preserved.

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

There were no dependencies between some loads and stores so they could be moved by each other.
Compiler Perspectives on Code Movement

- Compilers must also preserve *control dependence*
- Example
  
  ```
  if (c1)
    I1;
  if (c2)
    I2;
  ```

  I1 is control dependent on c1 and I2 is control dependent on c2 but not on c1.

  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.

Code Motion

- Can be done in SW or HW
- Why SW?
- Why HW?

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
HW Schemes: Instruction Parallelism

- Why in HW at run time?
  - Works when can’t know dependence at run time
  - Compiler simpler
  - Code for one machine runs well on another
- Key idea: Allow instructions behind stall to proceed

```
DIVD  F0, F2, F4
ADDD  F10, F0, F8
SUBD  F12, F8, F14
```
- Enables out-of-order execution => out-of-order completion
- ID stage checked both for structural & data dependencies

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

```
DIVD  F0,F2,F4 (10 cycles)
ADDD  F10, F0, F8 (4 cycles)
SUBD  F12, F8, F14
ADDD  F20,F2,F3
MULTD F13,F12,F2 (6 cycles)
ADDD  F4,F1,F3
ADDD  F5,F4,F13
```
(assume several FP ADD units)

CDC 6600 Scoreboard

- Enables dynamic scheduling
- Allows instructions to proceed when dependencies are satisfied
Scoreboard Implications

- Out-of-order completion => WAR, WAW hazards?
- Solution for WAR – stall WB until earlier (in program order) instruction reads operands.
- For WAW, must detect hazard: stall in ID until other completes

DIVD F0,F2,F4
ADDD F10,F0,F8
SUBD F8,F8,F14

Scoreboard Implications, cont.

- Need to have multiple instructions in execution phase => multiple execution units or pipelined execution units
- Scoreboard keeps track of dependencies, state or operations
- Scoreboard replaces ID, EX, WB with 4 stages

Four Stages of Scoreboard Control

1. **Issue**—decode instructions & check for structural hazards (ID1)
   If a functional unit for the instruction is free and no other active instruction has the same destination register (WAW), the scoreboard issues the instruction to the functional unit and updates its internal data structure. If a structural or WAW hazard exists, then the instruction issue stalls, and no further instructions will issue until these hazards are cleared.

2. **Read operands**—wait until no data hazards, then read operands (ID2)
   A source operand is available if no earlier issued active instruction is going to write it, or if the register containing the operand is being written by a currently active functional unit. When the source operands are available, the scoreboard tells the functional unit to proceed to read the operands from the registers and begin execution. The scoreboard resolves RAW hazards dynamically in this step, and instructions may be sent into execution out of order.

Example:

```
DIVD F0,F2,F4
ADDD F10,F0,F8
SUBD F8,F8,F14
```

CDC 6600 scoreboard would stall SUBD until ADDD reads operands

Four Stages of Scoreboard Control

3. **Execution**—operate on operands (EX)
   The functional unit begins execution upon receiving operands. When the result is ready, it notifies the scoreboard that it has completed execution.

4. **Write result**—finish execution (WB)
   Once the scoreboard is aware that the functional unit has completed execution, the scoreboard checks for WAR hazards. If none, it writes results. If WAR, then it stalls the instruction.

Example:

```
DIVD F0,F2,F4
ADDD F10,F0,F8
SUBD F8,F8,F14
```

CDC 6600 scoreboard would stall SUBD until ADDD reads operands
Three Parts of the Scoreboard

1. **Instruction status**—which of 4 steps the instruction is in

2. **Functional unit status**—Indicates the state of the functional unit (FU). 9 fields for each functional unit
   - Busy—Indicates whether the unit is busy or not
   - Op—Operation to perform in the unit (e.g., + or –)
   - Fi—Destination register
   - Fj, Fk—Source-register numbers
   - Qj, Qk—Functional units producing source registers Fj, Fk
   - Rj, Rk—Flags indicating when Fj, Fk are ready

3. **Register result status**—Indicates which functional unit will write each register, if one exists. Blank when no pending instructions will write that register