Impact of instruction ordering

- Pre-1982: microprocessors ran instructions implemented in microcode instructions
  - Memory faster than processor; always 1 cycle to access
  - Time to execute instruction sequence = sum of individual instruction times
- Modern processors (MIPS, ≥ 80486)
  - Pipelining, multiple functional units allow different instruction executions to overlap -- different orderings produce varying degrees of overlap
  - Memory may take ~100 cycles to access: loads should be started as early as possible
- Instruction order has significant performance impact on modern architectures

Instruction ordering issues

- Modern superscalar architecture "executes N instructions every cycle"
- Pentium: N = 2 (U-pipe and V-pipe)
- Pentium II+: N = 5; dynamic translation to 1-4+ µops
- Reality check: about 1.2 instructions per cycle on average with good instruction ordering -- processor resources are usually wasted
- Processor spends a lot of time waiting:
  - Branch stalls
  - Memory stalls
  - Expensive arithmetic operations
- Avoiding stalls requires understanding processor architecture(s) (Intel Arch. SDM Vol. 3, Chapter 13)

Simplified architecture model

- Assume simple MIPS-like pipelined architecture -- 5 pipeline stages (Pentium II: 9)
- F: Instruction fetch -- read instruction from memory, decode
- R: Read values from registers
- A: ALU
- M: Memory load or store
- W: Write back result to registers

Examples

- mov ax, bx
  - F: read bx
  - W: store into ax
- add ax, 10
  - F: extract imm. 10
  - R: read ax
  - A: add operands
  - W: store into ax
- mov cx, [dx + 16]
  - R: read dx, 16
  - A: compute address
  - M: read from cache
  - W: store into cx
- push [dx + 16]?
Non-pipelined execution

- **add ax, 10**
- **mov cx, [dx + 16]**
- **3-5 cycles per instruction**

Pipelined Execution

- **New instruction begun every cycle**
- **Most pipeline stages busy every cycle**

Superscalar execution

- **Two copies of execution units**
- **Many instructions executed in parallel**

Solutions:

- **Option 1:** (original Alpha, 486, Pentium)
  Processor stalls on use of result until available. Compiler should **reorder** instructions if possible:
  - `mov ax, [cx + 16]`
  - `add bx, ax`
  - `add cx, 1`  
  - `mov ax, [cx + 16]`
  - `add bx, ax`

- **Option 2:** (R3000) Memory result not available until two instructions later; compiler **must** insert some instruction.
  - `mov ax, [cx + 16]`
  - `jmp`  
  - `mov bx, ax`
  - `add cx, 1`  
  - `mov ax, [cx + 16]`
  - `add cx, 1`
  - `mov bx, ax`
Out-of-order execution

- Out-of-order execution (PowerPC, recent Alpha, MIPS, P6): can execute instructions further ahead rather than stall – compiler instruction ordering is less important
- Processor has reorder buffer from which viable instructions are selected on each cycle

\[
\begin{align*}
&\text{mov ax, [cx + 16]} \quad F \ R \ A \ M \ W \\
&\text{mov bx, ax} \quad F \ R \ A \ M \ W \\
&\text{add cx, 1} \quad F \ R \ A \ M \ W \\
\end{align*}
\]

Branch stalls

- Branch, indirect jump instructions: next instruction to execute not known until address known
- Processor stall of 3-10 cycles!

\[
\begin{align*}
&\text{cmp ax, bx} \quad F \ R \ A \ M \ W \\
&\text{cmp ay, bx} \quad *F \ R \ A \ M \ W \\
&\text{beq r1, r2, L} \quad F \ R \ A \ M \ W
\end{align*}
\]

Option 1: stall

- 80486 stalls branches till pipeline empty!
- Early Alpha processors: start initial pipeline stages on predicted branch target, stall until target address known (3+ cycle stall on branch mispredict)

\[
\begin{align*}
&\text{beq r1, L} \quad F \ R \ A \ M \ W \\
&\text{mov r2, r3} \quad F \ R \ A \ M \ W \\
&\text{ld r4, [t6+16]} \quad F \ R \ A \ M \ W
\end{align*}
\]

Dealing with stalls

- Alpha: predicts backward branches taken (loops), forward branches not taken (else clauses), also has branch prediction cache
- Compiler should avoid branches, indirect jumps
  - unroll loops!
  - use conditional move instructions (Alpha, Pentium Pro) or predicated instructions (Itanium) – can be inserted as low-level optimization on assembly code

\[
\begin{align*}
&\text{cmp cx, 16} \quad F \ R \ A \ M \ W \\
&\text{cmovz ax, bx} \quad F \ R \ A \ M \ W \\
\end{align*}
\]

MIPS: branch delay slot

- Instruction after branch is always executed: branch delay slot

\[
\begin{align*}
&\text{beq r1, r2, L} \quad F \ R \ A \ M \ W \\
&\text{mov ax, bx} \quad F \ R \ A \ M \ W \\
&\text{move earlier instruction after branch} \\
&\text{move destination instruction if harmless} \\
&\text{Problem: branch delay slot hasn’t scaled}
\end{align*}
\]

Real architectures

- Deeper pipelines, superscalar
  - MIPS R4000: 8 stages; R10000: 8 stages x 4 way
  - Alpha: 11 stages, 2 or 4 way
  - Pentium P6: 9 stage (uops), 5 way, 9 stage dynamic translation pipeline
- Some instructions take much longer to complete - multiply, divide, cache miss
- Even register operands may not be ready in time for next instruction
**Resource conflicts**
- Typical superscalar processors: 4-way
- < 4 copies of some functional units
- R10000: 2 integer ALU units, 2 floating point ALU units. Pentium: 1/1
- Issuing too many ALU operations at once means some pipelines stall -- want to interleave other kinds of operations to allow all 4 pipelines to fill

**Instruction scheduling**
- Goal: reorder instructions so that all pipelines are as full as possible
- Instructions reordered against some particular machine architecture and scheduling rules embedded in hardware
- May need to compromise so that code works well on a variety of architectures (e.g. Pentium vs. Pentium II)

**Scheduling constraints**
- Instruction scheduling is a low-level optimization: performed on assembly code
- Reordered code must have same effect as original
- Constraints to be considered:
  - data dependencies
  - control dependencies: only within BB
  - resource constraints

**Data dependencies**
- If two instructions access the same register or memory location, they may be dependent
- True dependency: write/read
  
  ```
  mov ax, [cx + 16]; add bx, ax
  ```
- Anti-dependency: read/write
  
  ```
  add bx, ax; mov ax, [cx + 16]
  ```
- Output dependency: write/write
  
  ```
  mul bx; mov ax, cx - both update ax
  ```

**Example**
1. mov cx, [bp+8]
2. add cx, ax
3. mov [bp + 4], ax
4. mov dx, [cx + 4]
5. add ax, dx
6. mov [bp + 4], bx

**Dependency Graph**
- If one instruction depends on another, order cannot be reversed -- constrains scheduling
- Register dependencies easy to identify
- Memory dependencies are trickier: two memory addresses may be *aliases* for each other -- need *alias analysis*
  
  ```
  mov [dx + 16], ax
  ```
  
  dependency?
Simple reordering

- Reorder only within basic block
- Construct dependence graph for each basic block
  - nodes are instructions
  - edges are instruction dependencies
  - graph will be a DAG (no cycles)
- Any valid ordering must make all dependence edges go forward in code: topological sort of dependence graph

List Scheduling Algorithm

- Initialize ready list R with all instructions not dependent on any other instruction
- Loop until R is empty
  - pick best node in R and append it to reordered instructions
  - update ready list with ready successors to best node
- Works for simple & superscalar processors
- Problem: Determining best node in R is NP-complete! Must use heuristic.

Greedy Heuristic

- If instruction predecessors won’t be sufficiently complete yet, creates stall
- Choose instruction that will be scheduled as soon as possible, based on start time of its predecessors: simulate processor
- How to break ties:
  - pick node with longest path to DAG leaf
  - pick node that can go to non-busy pipeline
  - pick node with many dependent successors

Scheduling w/ FRAMW model

<table>
<thead>
<tr>
<th>Ready</th>
<th>1,3</th>
<th>2,3</th>
<th>2,6</th>
<th>4,6</th>
<th>5,6</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1FRAMW</td>
<td>2FRAMW</td>
<td>3FRAMW</td>
<td>4FRAMW</td>
<td>5FRAMW</td>
<td>5FRAMW</td>
</tr>
</tbody>
</table>

Result: eliminated stalls after 1 & 4, moved memory operations earlier

Register allocation conflict

- Problem: use of same register creates anti-dependencies that restrict scheduling
- Register allocation before scheduling: prevents good scheduling
- Scheduling before register allocation: spills destroy scheduling
- Solution: schedule abstract assembly, allocate registers, schedule again!

Summary

- Instruction scheduling very important for non-fancy processors
- Improves performance even on processors with out-of-order execution (dynamic reordering must be more conservative)
- List scheduling provides a simple heuristic for instruction scheduling