How to improve (decrease) CPI

• Recall: \( \text{CPI} = \text{Ideal CPI} + \text{CPI contributed by stalls} \)
• Ideal CPI =1 for single issue machine even with multiple pipes
• Ideal CPI will be less than 1 if we have several pipes and we can issue (and “commit”) multiple instructions in the same cycle, i.e., we take advantage of Instruction Level Parallelism (ILP)

Exploitation of Instruction Level Parallelism (ILP)

• Will increase throughput and decrease CPU execution time
• Will increase structural hazards
  – Cannot issue 2 instructions to the same pipe
• Makes reduction in other stalls even more important
  – A stall costs more than the loss of a single instruction issue
• Will make the design more complex
  – WAW and WAR hazards can occur
  – Out-of-order completion can occur
  – Precise exception handling is more difficult

Where can we optimize? (control)

• CPI contributed by control stalls can be decreased statically (compiler) or dynamically (hardware)
• Compiler optimizations
  – Reduce the number of branches: loop unrolling
• Speculative execution
  – Static (software) or dynamic (hardware) branch prediction
  – Code movement: execute instructions before knowing that they will need to be executed (beware of exceptions)
  • Speculative loads
  – Predication

Where can we optimize? (data dependencies)

• CPI contributed by data hazards can be decreased statically (compiler) or dynamically (hardware)
• Compiler optimizations
  – Load scheduling, dependence analysis, software pipelining, trace scheduling
• Hardware (run-time) techniques
  – Forwarding (RAW)
  – Register renaming (WAW, WAR)

Data dependencies (RAW)

• Instruction (statement) \( S_j \) dependent on \( S_i \) if
  \[ O_j \cap I_i = \emptyset \]
  – Transitivity: Instruction \( j \) dependent on \( k \) and \( k \) dependent on \( i \)
• Dependence is a program property
• Hazards (RAW in this case) and their (partial) removals are a pipeline organization property
• Code scheduling goal
  – Maintain dependence and avoid hazard (pipeline is exposed to the compiler)

Loop unrolling

• Pros
  – Decrease loop overhead (branches, counter settings)
  – Allows better scheduling
  • Longer basic blocks hence better opportunities to hide latency of “long” operations and to prevent load delays
• Cons
  – Increases register pressure
  – Increases code length (I-cache occupancy)
  – Requires prologue or epilogue
Software pipelining

- Reorganize loops with loop-carried dependences by “symbolically” unrolling them
  - New code: statements of distinct iterations of original code
  - Take an “horizontal” slice of several (dependent) iterations

![Software pipelining diagram]

Name dependence

- **Anti dependence** $O_i \cap I_j \neq \emptyset$
  - $S_i: R1 \leftarrow R1+R2; \ldots; S_j: R1 \leftarrow \ldots$
  - At the instruction level, this is a WAR hazard if instruction $j$ finishes first

- **Output dependence** $O_i \cap O_j \neq \emptyset$
  - $S_i: R1 \leftarrow \ldots; S_j: R1 \leftarrow \ldots$
  - At the instruction level, this is a WAW hazard if instruction $j$ finishes first

- In both cases, not really a dependence but a “naming” problem
  - Register renaming (compiler by register allocation, in hardware see later)

Control dependencies

- Branches restrict the scheduling of instructions
- **Speculation** (i.e., executing an instruction that might not be needed) must be:
  - Safe (no additional exception)
  - Legal (the end result should be the same as without speculation)
- Speculation can be implemented by, for example:
  - Compiler (code motion)
  - Hardware (branch prediction)
  - Both (predication)