Outline

- ILP
- Compiler techniques to increase ILP
- Loop Unrolling
- Static Branch Prediction
- Dynamic Branch Prediction
- Overcoming Data Hazards with Dynamic Scheduling
- Tomasulo Algorithm
- Conclusion

Recall from Pipelining

- Pipeline CPI = Ideal pipeline CPI + Structural Stalls + Data Hazard Stalls + Control Stalls
  - Ideal pipeline CPI: measure of the maximum performance attainable by the implementation
  - Structural hazards: HW cannot support this combination of instructions
  - Data hazards: Instruction depends on result of prior instruction still in the pipeline
  - Control hazards: Caused by delay between the fetching of instructions and decisions about changes in control flow (branches and jumps)

Instruction-Level Parallelism

- Instruction-Level Parallelism (ILP)
  - Overlap the execution of instructions to improve performance
  - 2 approaches to exploit ILP
    1. Rely on hardware to help discover and exploit the parallelism dynamically
       - Pentium 4, AMD Opteron, IBM Power
    2. Rely on software technology to find parallelism, statically at compile-time
       - Itanium 2 / IA-64

Instruction-Level Parallelism (ILP)

- Basic Block (BB) ILP is quite small
  - BB: a straight-line code sequence with no branches in except to the entry and no branches out except at the exit
  - average dynamic branch frequency 15% to 25%
    => 4 to 7 instructions execute between a pair of branches
  - Plus instructions in BB likely to depend on each other
  - Need ILP across multiple basic blocks

Loop-Level Parallelism

- Simplest: loop-level parallelism to exploit parallelism among iterations of a loop.
  - Example
    for (i=1; i<=1000; i=i+1)
    x[i] = x[i] + y[i];
  - Exploit loop-level parallelism by “unrolling loop” either by
    - dynamic via branch prediction or
    - static via loop unrolling by compiler
    (Another way is vectors, to be covered later)
Loop-Level Parallelism

- Determining dependences critical
- If 2 instructions are
  - parallel, they can execute simultaneously in a pipeline of arbitrary depth without causing any stalls (assuming no structural hazards)
  - dependent, they are not parallel and must be executed in order, although they may often be partially overlapped

Data Dependence and Hazards

- Instr \_j is data dependent (aka true dependence) on Instr \_i:
  1. Instr \_j tries to read operand before Instr \_i writes it
  2. or Instr \_j reads operand, which is dependent on Instr \_l, which is dependent on Instr \_i
- If two instructions are data dependent, they cannot execute simultaneously or be completely overlapped
- Data dependence in instruction sequence = data dependence in source code
  => effect of original data dependence must be preserved
- If data dependence caused a hazard in pipeline, that’s a Read After Write (RAW) hazard

ILP and Data Dependencies, Hazards

- HW/SW must preserve illusion of program order: order instructions would execute in if executed sequentially as determined by original source program
  - dependences are a property of programs
- Presence of dependence indicates potential for a hazard, but
  - actual hazard and length of any stall is property of the pipeline
- Importance of the data dependencies
  1) indicates the possibility of a hazard
  2) determines order in which results must be calculated
  3) sets an upper bound on how much parallelism can possibly be exploited
- HW/SW goal: exploit parallelism by preserving program order only where it affects the outcome of the program

Name Dependence #1: Anti-dependence

- Name dependence: when 2 instructions use same register or memory location, called a name, but no flow of data between the instructions associated with that name; 2 versions of name dependence
- Instr \_j writes operand before Instr \_i reads it
  1. Instr \_i: sub r4, r1, r3
  2. Instr \_j: add r2, r1, r3
  3. Instr \_k: mul r6, r1, r7
  Called an "anti-dependence" by compiler writers.
  This results from reuse of the name "r1"
- If anti-dependence caused a hazard in the pipeline, that’s a Write After Read (WAR) hazard

Name Dependence #2: Output dependence

- Instr \_j writes operand before Instr \_i writes it.
  1. Instr \_i: sub r1, r4, r3
  2. Instr \_j: add r1, r2, r3
  3. Instr \_k: mul r6, r1, r7
  Called an "output dependence" by compiler writers
  This also results from the reuse of the name "r1"
- If anti-dependence caused a hazard in the pipeline, that’s a Write After Write (WAW) hazard
- Instructions involved in a name dependence can execute simultaneously if name used in instructions is changed so instructions do not conflict
  - Register renaming resolves name dependence for registers
  - Either by compiler or by HW

Control Dependencies

- Every instruction is control dependent on some set of branches, and, in general, these control dependencies must be preserved to preserve program order
  1. if p1 {
     S1;
   }
  2. if p2 {
     S2;
   }
  S1 is control dependent on p1, and S2 is control dependent on p2 but not on p1.
Control Dependence Ignored

- Control dependence need not be preserved
  - willing to execute instructions that should not have been executed, thereby violating the control dependences, if can do so without affecting correctness of the program
- Instead, 2 properties critical to program correctness are
  - exception behavior and
  - data flow

Exception Behavior

- Preserving exception behavior
  - any changes in instruction execution order must not change how exceptions are raised in program (no new exceptions)
- Example:
  - DADDU R2, R3, R4
  - BEQZ R2, L1
  - LW R1, 0 (R2)
  - L1:
  - (Assume branches not delayed)
- Problem with moving LW before BEQZ?

Data Flow

- Data flow: actual flow of data values among instructions that produce results and those that consume them
  - branches make flow dynamic, determine which instruction is supplier of data
- Example:
  - DADDU R1, R2, R3
  - BEQZ R4, L
  - DSUBU R1, R5, R6
  - L:
  - OR R7, R1, R8
- OR depends on DADDU or DSUBU?
  - Must preserve data flow on execution