Chapter 2
Instruction Level Parallelism

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 (e.g., Pentium 4, AMD Opteron, IBM Power), and
  2) Rely on software technology to find parallelism, statically at compile-time (e.g., Itanium 2)
- Next 4 lectures on this topic
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

- To obtain substantial performance enhancements, we must exploit ILP across multiple basic blocks

- Simplest: loop-level parallelism to exploit parallelism among iterations of a loop. E.g., for \((i=1; i<=1000; i=i+1)\)
  \[ x[i] = x[i] + y[i]; \]

Loop-Level Parallelism

- Exploit loop-level parallelism to 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)

- Determining instruction dependence is critical to Loop Level Parallelism

- 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
Types of Dependences

- Data Dependence (AKA “True dependence”)
- Name Dependence
- Control Dependence

Data Dependence and Hazards

- Instr$_j$ is data dependent on Instr$_i$ if:
  - Instr$_j$ reads an operand that Instr$_i$ writes
    
    $I$: add $r1, r2, r3$
    $J$: sub $r4, r1, r3$

  - or Instr$_j$ is data dependent on Instr$_k$ 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 $\Rightarrow$ data dependence in source code $\Rightarrow$ effect of original data dependence must be preserved
- If data dependence caused a hazard in pipeline, this is a Read After Write (RAW) hazard

Adapted from Patterson and Hennessy (Morgan Kaufman Pubs)
ILP and Data Dependences

• HW/SW must preserve program order: order instructions would execute in if executed sequentially as determined by original source program

• 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

• 2 instructions use same register or memory location (called a name) but there is no flow of data intended between the instructions associated with that name
Name Dependence #1: Anti-dependence

- Instr\textsubscript{j} writes operand \textit{before} Instr\textsubscript{i} reads it
  
  I: sub r4, r1, r3  
  J: add r1, r2, r3  
  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, called a Write After Read (WAR) hazard

- WAR doesn’t happen in most pipelines of the types we’ve seen so far because they have reads early (in ID) and writes late (in WB).

Name Dependence #2: Output dependence

- Instr\textsubscript{j} writes operand \textit{before} Instr\textsubscript{i} writes it.
  
  I: sub r1, r4, r3  
  J: add r1, r2, r3  
  K: mul r6, r1, r7

  Called an “output dependence” by compiler writers
  This also results from the reuse of name “r1”

- If anti-dependence caused a hazard in the pipeline, called a Write After Write (WAW) hazard
Resolving Name Dependences

• 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 regs
  – Either by compiler or by HW

```
I: sub r4, r1, r3
J: add r1, r2, r3
K: mul r6, r1, r7
```

```
I: sub r4, r1, r3
J: add r5, r2, r3
K: mul r6, r1, r7
```

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
  ```
  if p1 {
    S1;
  }
  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
  - We are willing to execute instructions that should not have been executed, thereby violating the control dependences, if we can do so without affecting correctness of the program.

- Instead, 2 properties critical to program correctness are
  1) exception behavior and
  2) 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)

- If we move LW before BEQZ, we might have a (new) memory address exception
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

Adapted from Patterson and Hennessey
(Morgan Kaufman Pubs)