Chapter 3

Instruction-Level Parallelism and Its Exploitation
Introduction

- Pipelining become a universal technique in 1985
  - Overlaps execution of instructions
  - Exploits “Instruction Level Parallelism”

- Beyond this, there are two main approaches:
  - Hardware-based dynamic approaches
    - Used in server and desktop processors
    - Not used as extensively in PMP processors
  - Compiler-based static approaches
    - Not as successful outside of scientific applications
Instruction-Level Parallelism

- When exploiting instruction-level parallelism, goal is to maximize CPI
  - Pipeline CPI =
    - Ideal pipeline CPI +
    - Structural stalls +
    - Data hazard stalls +
    - Control stalls

- Parallelism with basic block is limited
  - Typical size of basic block = 3-6 instructions
  - Must optimize across branches (i.e., inter-block)
Data Dependence

- Loop-Level Parallelism
  - Unroll loop statically or dynamically
  - Use SIMD (vector processors and GPUs)

- Challenges:
  - Data dependency
    - Instruction \( j \) is data dependent on instruction \( i \) if
      - Instruction \( i \) produces a result that may be used by instruction \( j \)
      - Instruction \( j \) is data dependent on instruction \( k \) and instruction \( k \) is data dependent on instruction \( i \)

- Dependent instructions cannot be executed simultaneously
Data Dependence

- Dependencies are a property of programs
- Pipeline organization determines if dependence is detected and if it causes a stall
- Data dependence conveys:
  - Possibility of a hazard
  - Order in which results must be calculated
  - Upper bound on exploitable instruction level parallelism
- Dependencies that flow through memory locations are difficult to detect
  - E.g., 100(R4) and 20(R6) may point to the same memory location.
Name Dependence

- Two instructions use the same name but no flow of information
  - Not a true data dependence, *but is a problem when reordering instructions*
  - **Antidependence**: instruction j writes a register or memory location that instruction i reads
    - Initial ordering (i before j) must be preserved
  - **Output dependence**: instruction i and instruction j write the same register or memory location
    - Ordering must be preserved

- To resolve, use renaming techniques
Other Factors

Data Hazards
- Read after write (RAW) – data dependence
- Write after write (WAW) – output dependence
- Write after read (WAR) - antidependence

Control Dependence
- Ordering of instruction i with respect to a branch instruction
  - Instruction control dependent on a branch cannot be moved before the branch so that its execution is no longer controlled by the branch
  - An instruction not control dependent on a branch cannot be moved after the branch so that its execution is controlled by the branch
Examples

- **Example 1:**
  DADDU R1, R2, R3
  BEQZ R4, L
  DSUBU R1, R1, R6
  L: ...
  OR R7, R1, R8

- **Example 2:**
  DADDU R1, R2, R3
  BEQZ R12, skip
  DSUBU R4, R5, R6
  DADDU R5, R4, R9
  skip:
  OR R7, R8, R9

- OR instruction dependent on DADDU and DSUBU (see R1)
- Must preserve control dependence
- Assume R4 isn’t used after skip
  - Possible to move DSUBU before the branch
Compiler Techniques for Exposing ILP

- **Pipeline scheduling**
  - Separate dependent instruction from the source instruction by the pipeline latency of the source instruction

- **Example:**
  ```
  for (i=999; i>=0; i=i-1)
    x[i] = x[i] + s;
  ```

<table>
<thead>
<tr>
<th>Instruction producing result</th>
<th>Instruction using result</th>
<th>Latency in clock cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP ALU op</td>
<td>Another FP ALU op</td>
<td>3</td>
</tr>
<tr>
<td>FP ALU op</td>
<td>Store double</td>
<td>2</td>
</tr>
<tr>
<td>Load double</td>
<td>FP ALU op</td>
<td>1</td>
</tr>
<tr>
<td>Load double</td>
<td>Store double</td>
<td>0</td>
</tr>
</tbody>
</table>
Pipeline Stalls

Loop: 
L.D F0,0(R1)  
stall  
ADD.D F4,F0,F2  
stall  
stall  
S.D F4,0(R1)  
DADDUI R1,R1,#-8  
stall (assume integer load latency is 1)  
BNE R1,R2,Loop  

<table>
<thead>
<tr>
<th>Instruction producing result</th>
<th>Instruction using result</th>
<th>Latency in clock cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP ALU op</td>
<td>Another FP ALU op</td>
<td>3</td>
</tr>
<tr>
<td>FP ALU op</td>
<td>Store double</td>
<td>2</td>
</tr>
<tr>
<td>Load double</td>
<td>FP ALU op</td>
<td>1</td>
</tr>
<tr>
<td>Load double</td>
<td>Store double</td>
<td>0</td>
</tr>
</tbody>
</table>
Pipeline Scheduling

Scheduled code:
Loop:      L.D    F0,0(R1)
           DADDUI R1,R1,#-8
           ADD.D F4,F0,F2
           stall
           stall
           S.D F4,8(R1)
           BNE R1,R2,Loop

<table>
<thead>
<tr>
<th>Instruction producing result</th>
<th>Instruction using result</th>
<th>Latency in clock cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP ALU op</td>
<td>Another FP ALU op</td>
<td>3</td>
</tr>
<tr>
<td>FP ALU op</td>
<td>Store double</td>
<td>2</td>
</tr>
<tr>
<td>Load double</td>
<td>FP ALU op</td>
<td>1</td>
</tr>
<tr>
<td>Load double</td>
<td>Store double</td>
<td>0</td>
</tr>
</tbody>
</table>
Loop Unrolling

- Loop unrolling
  - Unroll by a factor of 4 (assume # elements is divisible by 4)
  - Eliminate unnecessary instructions

Loop:

L.D F0,0(R1)
ADD.D F4,F0,F2
S.D F4,0(R1) ;drop DADDUI & BNE
L.D F6,-8(R1)
ADD.D F8,F6,F2
S.D F8,-8(R1) ;drop DADDUI & BNE
L.D F10,-16(R1)
ADD.D F12,F10,F2
S.D F12,-16(R1) ;drop DADDUI & BNE
L.D F14,-24(R1)
ADD.D F16,F14,F2
S.D F16,-24(R1)
DADDUI R1,R1,#-32
BNE R1,R2,Loop

- note: number of live registers vs. original loop
Loop Unrolling/Pipeline Scheduling

- Pipeline schedule the unrolled loop:

  Loop:  
  L.D F0,0(R1) 
  L.D F6,-8(R1) 
  L.D F10,-16(R1) 
  L.D F14,-24(R1) 
  ADD.D F4,F0,F2 
  ADD.D F8,F6,F2 
  ADD.D F12,F10,F2 
  ADD.D F16,F14,F2 
  S.D F4,0(R1) 
  S.D F8,-8(R1) 
  DADDUI R1,R1,#-32 
  S.D F12,16(R1) 
  S.D F16,8(R1) 
  BNE R1,R2,Loop
Dynamic Scheduling

- Rearrange order of instructions to reduce stalls while maintaining data flow

Advantages:
- Compiler doesn’t need to have knowledge of microarchitecture
- Handles cases where dependencies are unknown at compile time

Disadvantage:
- Substantial increase in hardware complexity
- Complicates exceptions
Dynamic Scheduling

- Dynamic scheduling implies:
  - Out-of-order execution
  - Out-of-order completion

- Creates the possibility for WAR and WAW hazards

- Tomasulo’s Approach
  - Tracks when operands are available
  - Introduces register renaming in hardware
    - Minimizes WAW and WAR hazards
Register Renaming

- Example:

  DIV.D F0,F2,F4
  ADD.D F6,F0,F8
  S.D F6,0(R1)
  SUB.D F8,F10,F14
  MUL.D F6,F10,F8

  + name dependence with F6

  Antidependence (WAR on F8)
  Antidependence (WAR on F6)
Register Renaming

- Example:

  DIV.D F0,F2,F4
  ADD.D S,F0,F8
  S.D S,0(R1)
  SUB.D T,F10,F14
  MUL.D F6,F10,T

- Now only RAW hazards remain, which can be strictly ordered
Multiple Issue and Static Scheduling

- To achieve CPI < 1, need to complete multiple instructions per clock

- Solutions:
  - Statically scheduled superscalar processors
  - VLIW (very long instruction word) processors
  - Dynamically scheduled superscalar processors
# Multiple Issue and Static Scheduling

<table>
<thead>
<tr>
<th>Common name</th>
<th>Issue structure</th>
<th>Hazard detection</th>
<th>Scheduling</th>
<th>Distinguishing characteristic</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Superscalar (static)</td>
<td>Dynamic</td>
<td>Hardware</td>
<td>Static</td>
<td>In-order execution</td>
<td>Mostly in the embedded space: MIPS and ARM, including the ARM Coretex A8</td>
</tr>
<tr>
<td>Superscalar (dynamic)</td>
<td>Dynamic</td>
<td>Hardware</td>
<td>Dynamic</td>
<td>Some out-of-order execution, but no speculation</td>
<td>None at the present</td>
</tr>
<tr>
<td>Superscalar (speculative)</td>
<td>Dynamic</td>
<td>Hardware</td>
<td>Dynamic with speculation</td>
<td>Out-of-order execution with speculation</td>
<td>Intel Core i3, i5, i7; AMD Phenom; IBM Power 7</td>
</tr>
<tr>
<td>VLIW/LIW</td>
<td>Static</td>
<td>Primarily software</td>
<td>Static</td>
<td>All hazards determined and indicated by compiler (often implicitly)</td>
<td>Most examples are in signal processing, such as the TI C6x</td>
</tr>
<tr>
<td>EPIC</td>
<td>Primarily static</td>
<td>Primarily software</td>
<td>Mostly static</td>
<td>All hazards determined and indicated explicitly by the compiler</td>
<td>Itanium</td>
</tr>
</tbody>
</table>

*Note: This table outlines different scheduling and hazard detection methods for various processors.*
VLIW Processors

- Package multiple operations into one instruction

- Example VLIW processor:
  - One integer instruction (or branch)
  - Two independent floating-point operations
  - Two independent memory references

- Must be enough parallelism in code to fill the available slots
VLIW Processors

Disadvantages:
- Statically finding parallelism
- Code size
- No hazard detection hardware
- Binary code compatibility