Chapter 3 – Instruction-Level Parallelism and its Exploitation (Part 3)

ILP vs. Parallel Computers
Dynamic Scheduling (Section 3.4, 3.5)
Dynamic Branch Prediction (Section 3.3)
Hardware Speculation and Precise Interrupts (Section 3.6)
Multiple Issue (Section 3.7)
Static Techniques (Section 3.2, Appendix H)
Limitations of ILP (Section 3.10)
Multithreading (Section 3.12)
Putting it Together (Mini-projects)

Beyond Pipelining

Limits on Pipelining
- Latch overheads & signal skew
- Unpipelined instruction issue logic (Flynn limit: CPI \( \geq 1 \))
Two techniques for parallelism in instruction issue
- Superscalar or multiple issue
  - Hardware determines which of next \( n \) instructions can issue in parallel
  - Maybe statically or dynamically scheduled
- VLIW – Very Long Instruction Word
  - Compiler packs multiple independent operations into an instruction

Simple 5-Stage Superscalar Pipeline

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i+1</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i+2</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i+3</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i+4</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i+5</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i+6</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i+7</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i+8</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i+9</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Superscalar, cont.

- IF: Parallel access to I-cache
  - Require alignment?
- ID: Replicate logic
  - Fixed-length instructions?
  - HANDLE INTRA-CYCLE HAZARDS
- EX: Parallel/pipelined (as before)
  - If so, hazards & multi-ported D-cache
- MEM: > 1 per cycle?
  - Different register files?
  - Multi-ported register files?
- WB: Progression:
  - Integer + floating-point
  - Any two instructions
  - Any four instructions
  - Any \( n \) instructions?
Assume two instructions per cycle
One integer, load/store, or branch
One floating point
Could require 64-bit alignment and ordering of instruction pair.

<table>
<thead>
<tr>
<th>I</th>
<th>F</th>
<th>I</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>OK</td>
<td>NOT</td>
<td>NOT</td>
<td>OK</td>
</tr>
</tbody>
</table>

Best case
CPI = 0.5
But ....

Hazards are a big problem
Loads
Latency is 1 cycle
Was 1 instruction
NOW 3 instructions
Branches
NOW 3 instructions
Floating point loads and stores
May cause structural hazards
Additional ports?
Additional stalls?
Parallelism required =

VLIW = Very Long Instruction Word Processors
Static multiple issue
Compiler packs multiple independent operations into an instruction
Like horizontal microcode
Versus Superscalar

Inherent limitations of ILP
Difficulties in building hardware
Increase ports to registers
Increase ports to memory
Duplicate FUs
Decoding in superscalar and impact on clock rate
Limitations specific to VLIW
Code size, binary compatibility
Compiler Techniques to Expose ILP

Many compiler techniques exist
Several used for multiprocessors as well
Our focus on techniques specifically for ILP

Loop Unrolling

Add scalar to vector
Loop: L.D F0, 0(R1)
  stall
  ADD.D F4, F0, F2
  stall
  S.D 0(R1), F4
  DSUBUI R1, R1, #8
  stall
  BNEZ R1, Loop
  stall

With scheduling
Loop: L.D F0, 0(R1)
  DSUBUI R1, R1, #8
  ADD.D F4, F0, F2
  stall
  BNEZ R1, Loop ; Assume delayed branch
  S.D #8(R1), F4

Unrolling the loop
Loop: L.D F0, 0(R1)
  ADD.D F4, F0, F2
  S.D 0(R1), F4
  L.D F6, -8(R1)
  ADD.D F8, F6, F2
  S.D -8(R1), F8
  L.D F10, -16(R1)
  ADD.D F12, F10, F2
  S.D -16(R1), F12
  L.D F14, -24(R1)
  ADD.D F16, F14, F2
  S.D -24(R1), F16
  DSUBUI R1, R1, #32
  BNEZ R1, Loop ; Assume delayed branch

Rename registers
Remove some branch overhead (calculate intermediate values)

Loop Unrolling

Scheduling the loop for simple pipeline
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 8(R1), F4
  S.D -8(R1), F8
  S.D -16(R1), F12
  DSUBUI R1, R1, #32
  BNEZ R1, Loop ; Assume delayed branch
  S.D 8(R1), F16

How to schedule for multiple issue?
Software Pipelining

Pipeline loops in software
Pipeline loop iteration
Executes instructions from multiple iterations of original loop
Separates dependent instructions
Less code than unrolling

Software Pipelining – Example

```
sum = 0.0;
for (i=1; i<=N; i++) {
    load a[i]; Ai
    load b[i]; Bi
    mult ab[i]; *i
    add sum[i]; +i
}
```

```
sum = 0.0;
START-UP-BLOCK
for (i=3; i<=N; i++) {
    load a[i]; Ai
    load b[i]; Bi
    mult ab[i-1]; *i-1
    add sum[i-2]; +i-2
}
FINISH-UP-BLOCK
```

Global Scheduling

Loop unrolling and software pipelining work well for straightline code
What if code has branches?
Global scheduling techniques
Trace scheduling

Trace Scheduling

Compiler predicts most frequently executed execution path (trace)
Schedules this path and inserts repair code for mispredictions
**Trace Scheduling - Example**

\[ b(i) = \text{"old"} \]
\[ a(i) = \]
if \[ a(i) == 0 \] then
\[ b(i) = \text{"new"}; \text{ common case} \]
else
\[ X \]
endif
\[ c(i) = \]

Until done
Select most common path - a trace
Schedule trace across basic blocks
Repair other paths

<table>
<thead>
<tr>
<th>trace to be scheduled</th>
<th>repair code</th>
</tr>
</thead>
<tbody>
<tr>
<td>b(i) = \text{&quot;old&quot;}</td>
<td>A: restore old b(i)</td>
</tr>
<tr>
<td>a(i) = \</td>
<td>X</td>
</tr>
<tr>
<td>b(i) = \text{&quot;new&quot;}</td>
<td>maybe recalculate c[i]</td>
</tr>
<tr>
<td>c(i) = \</td>
<td>goto B</td>
</tr>
<tr>
<td>if [ a(i) != 0 ] goto A</td>
<td></td>
</tr>
<tr>
<td>B:</td>
<td></td>
</tr>
</tbody>
</table>

**Hardware Support to Expose Compile-Time ILP**

Compiler scheduling limited by knowledge of branch behavior

Hardware support to help compiler

- Predicated (or guarded or conditional) instructions
- Hardware support for compiler speculation

**Predicated Instructions**

Used to convert control dependence to data dependence

Instruction executed based on a predicate (or guard or condition)

If condition is false, then no result write or exceptions

**Predicated Instructions (Cont.)**

Example

\[
\begin{align*}
\text{if (condition) then } & \{ \\
& A = B; \\
& \}
\end{align*}
\]

... Convert to:

\[
\begin{align*}
R & \leftarrow \text{ result of condition evaluation} \\
A & \leftarrow B \text{ (predicated on } R) \\
\end{align*}
\]

Hardware can schedule instructions across the branch

Alpha, MIPS, PowerPC, SPARC V9, x86 (Pentium) have conditional moves

IA-64 has general predication - 64 1-bit predicate bits

Limitations

- Takes a clock even if annulled
Successful compiler scheduling requires
Preservation of exception behavior on speculation
Mechanism to speculatively reorder memory operations

What if there is an exception on a speculative instruction?
Distinguish between two classes of exceptions
(1) Indicate program error and require termination (e.g., protection violation)
(2) Can be handled and program resumed (e.g., page fault)
Type (2) can be handled immediately even for speculative instructions
Type (1) requires more support
Poison bits

Poison Bits
Hardware support
A poison bit for each register
A speculation bit for each instruction
If a speculative instruction sees an exception
it sets poison bit of destination
If a speculative instruction sees poison bit set for source
it propagates poison bit to its destination
If normal instruction sees poison bit for source, takes exception
Normal instruction resets poison bit of destination register

Hardware for Preserving Exception Behavior
Hardware for Memory Speculation
How to reorder memory ops if compiler is not sure of addresses?
Consider moving a load
Insert a special check instruction at original location of load
When load is executed, hardware saves its address
If there is a store to L’s address before the check instruction
Redo load
Branch to fix up code if other instructions already used load’s value