Lecture 6.3: Instructional Level Parallelism: Advanced Techniques

Marc Moreno Maza

www.csd.uwo.ca/Courses/CS3350b

[Adapted from lectures on Computer Organization and Design, Patterson & Hennessy, 5th edition, 2011]
Greater Instruction-Level Parallelism

- Deeper pipeline (more #stages: 5 => 10 => 15 stages)
  - Less work per stage ⇒ shorter clock cycle

- Multiple issue “superscalar”
  - Replicate pipeline stages ⇒ multiple pipelines
    - e.g., have two ALUs or a register file with 4 read ports and 2 write ports
    - have logic to issue several instructions concurrently
  - Execute more than one instruction at a clock cycle, producing an effective CPI < 1, so use *Instructions Per Cycle (IPC)*
  - e.g., 4GHz 4-way multiple-issue
    - 16 BIPS, peak CPI = 0.25, peak IPC = 4
  - If a datapath has a 5-stage pipeline, how many instructions are active in the pipeline at any given time?
  - But dependencies reduce this in practice
## Pipeline Depth and Issue Width

- Intel Processors over Time

<table>
<thead>
<tr>
<th>Microprocessor</th>
<th>Year</th>
<th>Clock Rate</th>
<th>Pipeline Stages</th>
<th>Issue width</th>
<th>Cores</th>
<th>Power</th>
</tr>
</thead>
<tbody>
<tr>
<td>i486</td>
<td>1989</td>
<td>25 MHz</td>
<td>5</td>
<td>1</td>
<td>1</td>
<td>5W</td>
</tr>
<tr>
<td>Pentium</td>
<td>1993</td>
<td>66 MHz</td>
<td>5</td>
<td>2</td>
<td>1</td>
<td>10W</td>
</tr>
<tr>
<td>Pentium Pro</td>
<td>1997</td>
<td>200 MHz</td>
<td>10</td>
<td>3</td>
<td>1</td>
<td>29W</td>
</tr>
<tr>
<td>P4 Willamette</td>
<td>2001</td>
<td>2000 MHz</td>
<td>22</td>
<td>3</td>
<td>1</td>
<td>75W</td>
</tr>
<tr>
<td>P4 Prescott</td>
<td>2004</td>
<td>3600 MHz</td>
<td>31</td>
<td>3</td>
<td>1</td>
<td>103W</td>
</tr>
<tr>
<td>Core 2 Conroe</td>
<td>2006</td>
<td>2930 MHz</td>
<td>14</td>
<td>4</td>
<td>2</td>
<td>75W</td>
</tr>
<tr>
<td>Core 2 Yorkfield</td>
<td>2008</td>
<td>2930 MHz</td>
<td>16</td>
<td>4</td>
<td>4</td>
<td>95W</td>
</tr>
<tr>
<td>Core i7 Gulftown</td>
<td>2010</td>
<td>3460 MHz</td>
<td>16</td>
<td>4</td>
<td>6</td>
<td>130W</td>
</tr>
</tbody>
</table>
Multiple-Issue Processor Styles

- **Static** multiple-issue processors, aka VLIW (very-long instruction word)
  - Decisions on which instructions to execute simultaneously are being made statically (at compile time by the compiler)
  - e.g. Intel Itanium and Itanium 2
    - 128-bit “bundles” containing three instructions
    - Five functional units (IntALU, Mmedia, Dmem, FPALU, Branch)
    - Extensive support for speculation and predication

- **Dynamic** multiple-issue processors (aka SuperScalar)
  - Decisions on which instructions to execute simultaneously (in the range of 2 to 8) are being made dynamically (at run time by the hardware)
    - e.g., IBM power series, Pentium 4, MIPS R10K, AMD Barcelona
Multiple-Issue Datapath Responsibilities

- Must handle, with a combination of hardware and software fixes, the fundamental limitations of
  - How many instructions to issue in one clock cycle – issue slots
  - Storage (data) dependencies – aka data hazards
    - Limitation more severe in a SS/VLIW processor due to (usually) low ILP
  - Procedural dependencies – aka control hazards
    - Ditto, but even more severe
    - Use dynamic branch prediction to help resolve the ILP issue
  - Resource conflicts – aka structural hazards
    - A SS/VLIW processor has a much larger number of potential resource conflicts
    - Functional units may have to arbitrate for result buses and register-file write ports
    - Resource conflicts can be eliminated by duplicating the resource or by pipelining the resource
Static Multiple Issue Machines (VLIW)

Static multiple-issue processors (aka VLIW) use the compiler (at compile-time) to statically decide which instructions to issue and execute simultaneously

- **Issue packet** – the set of instructions that are bundled together and issued in one clock cycle – think of it as one large instruction with multiple operations
- The mix of instructions in the packet (bundle) is usually restricted – a single "instruction" with several predefined fields
- The compiler does **static branch prediction** and **code scheduling** to reduce (control) or eliminate (data) hazards

VLIW’s have

- Multiple functional units
- Multi-ported register files
- Wide program bus
An Example: A VLIW MIPS

- Consider a 2-issue MIPS with a 2 instr bundle

Instructions are always fetched, decoded, and issued in pairs
- If one instr of the pair can not be used, it is replaced with a nop

- Need 4 read ports and 2 write ports and a separate memory address adder
Code Scheduling Example

- Consider the following loop code

```assembly
lp:    lw     $t0,0($s1)    # $t0=array element
       addu   $t0,$t0,$s2    # add scalar in $s2
       sw     $t0,0($s1)    # store result
       addi   $s1,$s1,-4     # decrement pointer
       bne    $s1,$0,lp      # branch if $s1 != 0
```

- Must “schedule” the instructions to avoid pipeline stalls
  - Instructions in one bundle must be independent
  - Must separate load/use instructions from their loads by one cycle
  - Notice that the first two instructions have a load/use dependency, the next two and last two have data dependencies
  - Assume branches are perfectly predicted by the hardware

/* increment each element (unsigned integer) in array A by n */
for (i=m; i>=0; --i)    /* m is the initial value of $s1 */
    A[i] += n;       /* n is the value in register $s2 */
### The Scheduled Code (Not Unrolled)

<table>
<thead>
<tr>
<th>ALU or branch</th>
<th>Data transfer</th>
<th>CC</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>lp:</strong> nop</td>
<td>lw $t0,0($s1)</td>
<td>1</td>
</tr>
<tr>
<td>addi $s1,$s1,−4</td>
<td>nop</td>
<td>2</td>
</tr>
<tr>
<td>addu $t0,$t0,$s2</td>
<td>nop</td>
<td>3</td>
</tr>
<tr>
<td>bne $s1,$0,lp</td>
<td>sw $t0,4($s1)</td>
<td>4</td>
</tr>
</tbody>
</table>

```
lp:    lw $t0,0($s1)     # $t0= array element
       addu $t0,$t0,$s2  # add scalar in $s2
       sw $t0,0($s1)     # store result
       addi $s1,$s1,−4   # decrement pointer
       bne $s1,$0,lp     # branch if $s1 != 0
```

**Four clock cycles to execute 5 instructions for a**

- CPI of 0.8 (versus the best case of 0.5?)
- IPC of 1.25 (versus the best case of 2.0?)
- noops don’t count towards performance!!
Loop Unrolling

- Loop unrolling – multiple copies of the loop body are made and instructions from different iterations are scheduled together as a way to increase ILP

- Apply loop unrolling (4 times for our example) and then **schedule** the resulting code
  - Eliminate unnecessary **loop overhead** instructions
  - Schedule so as to avoid **load use hazards**

- During unrolling the compiler applies **register renaming** to eliminate all data dependencies that are not true data dependencies
Loop Unrolling in C

```c
for (i=m; i>=0; --i)
    A[i] += n;
```

Assume size of A is 8, i.e. m=7.

Execute not-unrolled code:

<table>
<thead>
<tr>
<th>Iteration</th>
<th>i</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7</td>
<td>A[7] += n</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>A[1] += n</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>A[0] += n</td>
</tr>
</tbody>
</table>

Execute unrolled code:

```c
/* unrolled 4 times */
for (i=m; i>=0; i-=4){
    A[i]   += n;
    A[i-1] += n;
    A[i-2] += n;
    A[i-3] += n;
}
```

Iteration #1, i=7:
```
{ A[7] += n;
}
```

Iteration #2, i=3:
```
{ A[3] += n;
  A[1] += n;
  A[0] += n;
}```
Apply Loop Unrolling for 4 times

```assembly
lp:
  lw  $t0,0($s1)  # $t0=array element
  lw  $t1,-4($s1) # $t1=array element
  lw  $t2,-8($s1) # $t2=array element
  lw  $t3,-12($s1) # $t3=array element
  addu $t0,$t0,$s2  # add scalar in $s2
  addu $t1,$t1,$s2  # add scalar in $s2
  addu $t2,$t2,$s2  # add scalar in $s2
  addu $t3,$t3,$s2  # add scalar in $s2
  sw  $t0,0($s1)  # store result
  sw  $t1,-4($s1) # store result
  sw  $t2,-8($s1) # store result
  sw  $t3,-12($s1) # store result
  addi $s1,$s1,-16 # decrement pointer
  bne $s1,$0,lp  # branch if $s1 != 0

/* code in c */
for(i=m;i>=0;i-=4) {
  A[i]  += n;
  A[i-1] += n;
  A[i-2] += n;
  A[i-3] += n;
}
```

- Why not reuse $t0 but use $t1, $t2, $t3?
- Why -4,-8,-12 and $s1=$s1-16?
- How many times can a loop be unrolled?
Eight clock cycles to execute
14 instructions for a

- CPI of 0.57
  (versus the best case of 0.5)
- IPC of 1.8
  (versus the best case of 2.0)

/* code in c */
for(i=m;i>=0;i-=4)
{
    A[i]  += n;
    A[i-1] += n;
    A[i-2] += n;
    A[i-3] += n;
}
Summary of Compiler Support for VLIW Processors

- The compiler packs groups of **independent** instructions into the bundle
  - Done by **code re-ordering** (trace scheduling)

- The compiler uses **loop unrolling** to expose more ILP

- The compiler uses **register renaming** to solve name dependencies and ensures **no load use hazards** occur

- While superscalars use dynamic prediction, VLIW’s primarily depend on the compiler for **branch prediction**
  - Loop unrolling reduces the number of conditional branches
  - Predication eliminates if-then-else branch structures by replacing them with predicated instructions

- The compiler predicts **memory bank references** to help minimize memory bank conflicts
VLIW Advantages & Disadvantages

- Advantages
  - Simpler hardware (potentially less power hungry)
  - Potentially more scalable
    - Allow more instr’s per VLIW bundle and add more FUs

- Disadvantages
  - Programmer/compiler complexity and longer compilation times
    - Deep pipelines and long latencies can be confusing (making peak performance elusive)
  - Lock step operation, i.e., on hazard all future issues stall until hazard is resolved (hence need for predication)
  - Object (binary) code incompatibility
  - Needs lots of program memory bandwidth
  - Code bloat
    - Noops are a waste of program memory space
    - Loop unrolling to expose more ILP uses more program memory space
Dynamic Multiple Issue Machines (SS)

- Dynamic multiple-issue processors (aka SuperScalar) use hardware at run-time to dynamically decide which instructions to issue and execute simultaneously.

- Instruction-fetch and issue – fetch instructions, decode them, and issue them to a FU to await execution.

- Instruction-execution – as soon as the source operands and the FU are ready, the result can be calculated.

- Instruction-commit – when it is safe to, write back results to the RegFile or D$ (i.e., change the machine state).
Dynamic Multiple Issue Machines (SS)

Instruction fetch and decode unit

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Functional units

Integer

Integer

Integer

Integer

Floating point

Load-store

Commit unit

In-order issue

Out-of-order execute

In-order commit
Dynamic Pipeline Scheduling

- Allow the CPU to execute instructions **out of order** to avoid stalls
  - But commit result to registers in order

- Example
  
  ```
  lw $t0, 20($s2)
  addu $t1, $t0, $t2
  subu $s4, $s4, $t3
  slti $t5, $s4, 20
  ```

  - Can start **subu** while **addu** is waiting for **lw**
Why Do Dynamic Scheduling?

- Why not just let the compiler schedule code?
  - Disadvantages of compiler scheduling code

- Not all stalls are predictable
  - e.g., cache misses

- Can’t always schedule around branches
  - Branch outcome is dynamically determined

- Different implementations of an ISA have different latencies and hazards
Speculation

- “Guess” what to do with an instruction
  - Start operation as soon as possible
  - Check whether guess was right
    - If so, complete the operation
    - If not, roll-back and do the right thing

- Common to static and dynamic multiple issue

- Examples
  - Speculate on branch outcome (**Branch Prediction**)
    - Roll back if path taken is different
  - Speculate on **load**
    - Roll back if location is updated
# Out Of Order Intel

- All use OOO since 2001

<table>
<thead>
<tr>
<th>Microprocessor</th>
<th>Year</th>
<th>Clock Rate</th>
<th>Pipeline Stages</th>
<th>Issue width</th>
<th>Out-of-order/Speculation</th>
<th>Cores</th>
<th>Power</th>
</tr>
</thead>
<tbody>
<tr>
<td>i486</td>
<td>1989</td>
<td>25MHz</td>
<td>5</td>
<td>1</td>
<td>No</td>
<td>1</td>
<td>5W</td>
</tr>
<tr>
<td>Pentium</td>
<td>1993</td>
<td>66MHz</td>
<td>5</td>
<td>2</td>
<td>No</td>
<td>1</td>
<td>10W</td>
</tr>
<tr>
<td>Pentium Pro</td>
<td>1997</td>
<td>200MHz</td>
<td>10</td>
<td>3</td>
<td>Yes</td>
<td>1</td>
<td>29W</td>
</tr>
<tr>
<td>P4 Willamette</td>
<td>2001</td>
<td>2000MHz</td>
<td>22</td>
<td>3</td>
<td>Yes</td>
<td>1</td>
<td>75W</td>
</tr>
<tr>
<td>P4 Prescott</td>
<td>2004</td>
<td>3600MHz</td>
<td>31</td>
<td>3</td>
<td>Yes</td>
<td>1</td>
<td>103W</td>
</tr>
<tr>
<td>Core</td>
<td>2006</td>
<td>2930MHz</td>
<td>14</td>
<td>4</td>
<td>Yes</td>
<td>2</td>
<td>75W</td>
</tr>
<tr>
<td>Core 2 Yorkfield</td>
<td>2008</td>
<td>2930MHz</td>
<td>16</td>
<td>4</td>
<td>Yes</td>
<td>4</td>
<td>95W</td>
</tr>
<tr>
<td>Core i7 Gulftown</td>
<td>2010</td>
<td>3460MHz</td>
<td>16</td>
<td>4</td>
<td>Yes</td>
<td>6</td>
<td>130W</td>
</tr>
</tbody>
</table>
Streaming SIMD Extensions (SSE)

- SIMD: Single Instruction Multiple Data
- A data parallel architecture
- Both current AMD and Intel’s x86 processors have ISA and micro-architecture support SIMD operations
  - MMX, 3DNow!, SSE, SSE2, SSE3, SSE4, AVX
  - Many functional units
  - 8 128-bit vector registers: XMM0, XMM1, …, XMM7
  - See the flag field in /proc/cpuinfo

- SSE (Streaming SIMD extensions): a SIMD instruction set extension to the x86 architecture
  - Instructions for operating on multiple data simultaneously (vector operations): for (i=0; i<n; ++i) Z[i]=X[i]+Y[i];

- Programming SSE in C++: intrinsics
Does Multiple Issue Work?

- Yes, but not as much as we’d like
- Programs have real dependencies that limit ILP
- Some dependencies are hard to eliminate
  - e.g., pointer aliasing
- Some parallelism is hard to expose
  - Limited window size during instruction issue
- Memory delays and limited bandwidth
  - Hard to keep pipelines full
- Speculation can help if done well
Takeaway

- Pipelining is an important form of ILP
- Challenge is hazards
  - Forwarding helps with many data hazards
  - Delayed branch helps with control hazard in 5 stage pipeline
  - Load delay slot / interlock necessary
- More aggressive performance:
  - Longer pipelines
  - VLIW
  - Superscalar
  - Out-of-order execution
  - Speculation
  - SSE?