Modern processor design

Hung-Wei Tseng
Announcement

• Reading quizzes due tomorrow before class
• Final exam:
  • 9/5 (Fri.) 3p-6p here
  • Cumulative.
  • Consists of multiple choices (maybe), short answer questions, calculations, open-ended questions.
  • No open book, no cheat sheet allowed
  • Small pizza party afterwards
Outline

• Achieve CPI < 1
  – Improve instruction level parallelism
• SuperScalar
• Dynamic scheduling/Out-of-order execution
• Simultaneous multithreading
Instruction level parallelism
Let’s start from this code

LOOP: lw $t1, 0($a0)
    add $v0, $v0, $t1
    addi $a0, $a0, 4
    bne $a0, $t0, LOOP
lw   $t0, 0($sp)
lw   $t1, 4($sp)

If the current value of
$a0$ is $0x10000000$ and
$t0$ is $0x10001000$, what are the
dynamic instructions that the
processor will execute?
Pipelining

- Draw the pipeline execution diagram
  - assume that we have full data forwarding path
  - assume that we stall for control hazard

```
lw  $t1, 0($a0)
add $v0, $v0, $t1
addi $a0, $a0, 4
bne $a0, $t0, LOOP
lw  $t1, 0($a0)
add $v0, $v0, $t1
addi $a0, $a0, 4
bne $a0, $t0, LOOP
```

7 cycles per loop in average (if there are many iterations)
Instruction level parallelism

- We have used pipeline to shrink the cycle time as short as possible
- Pipeline increases the throughput by improving instruction level parallelism (ILP)
  - Instruction level parallelism: the processor can perform multiple instructions at the same cycle
- With data forwarding, branch prediction and caches, we still can only achieve CPI = 1 in the best case.
- Can we further improve ILP to achieve CPI < 1?
SuperScalar
Previously, we talked about pipeline SuperScalar!
SuperScalar

- Improve ILP by widen the pipeline
  - The processor can handle more than one instructions in one stage
  - Instead of fetching one instruction, we fetch multiple instructions!
- CPI = $\frac{1}{n}$ for an n-issue SS processor in the best case.

```plaintext
add  $t1, $a0, $a1
addi $a1, $a1, -1
add  $t2, $a0, $t1
bne  $a1, $zero, LOOP
add  $t1, $a0, $a1
addi $a1, $a1, -1
add  $t2, $a0, $t1
bne  $a1, $zero, LOOP
```

4 cycles per loop. 2 cycle per loop with perfect prediction. Pipeline takes 6 cycles per loop.
SuperScalar

- Improve ILP by widen the pipeline
  - The processor can handle more than one instructions in one stage
  - Instead of fetching one instruction, we fetch multiple instructions!
- CPI = 1/n for an n-issue SS processor in the best case.

 lw $t1, 0($a0)  
 add $v0, $v0, $t1  
 addi $a0, $a0, 4  
 bne $a0, $t0, LOOP  
 lw $t1, 0($a0)  
 add $v0, $v0, $t1  
 addi $a0, $a0, 4  
 bne $a0, $t0, LOOP

7 cycles per loop in worst case, 4 cycles if branch predictor predicts perfectly
Not very impressive...
Reordering instructions

Consider the following instructions:

1: lw $t1, 0($a0)
2: add $v0, $v0, $t1
3: addi $a0, $a0, 4
4: bne $a0, $t0, LOOP

Reordering which of the following pair of instructions would improve the performance running on top of a 2-issue SuperScalar processor without affecting correctness?

A. 1 and 3
B. 2 and 3
C. 2 and 4
D. 3 and 4
E. none of the above
Reordering using compiler

• We can use compiler optimization to reorder the instruction sequence
• Compiler optimization requires no hardware change

lw  $t1, 0($a0)
addi $a0, $a0, 4
add  $v0, $v0, $t1
bne  $a0, $t0, LOOP
lw  $t1, 0($a0)
addi $a0, $a0, 4
add  $v0, $v0, $t1
bne  $a0, $t0, LOOP

5 cycles per loop in worst case, 2 cycles if branch prediction perfectly
Very Long Instruction Word (VLIW)

- Each instruction word contains multiple instructions that can be executed concurrently
  - Compiler schedules the instructions
  - For an n-issue processor, each instruction word should contain n instructions.
  - Fill nops if cannot find n instructions to pack in an instruction word
- Benefit
  - Low power: no scheduling hardware required
- Real-world cases:
  - Itanium 2
  - AMD GPU
If we draw the data dependency graph of dynamic instructions

- Draw the data dependency graph, put an arrow if an instruction depends on the other.
- RAW (Read after write)

1: lw $t1, 0($a0)
2: add $v0, $v0, $t1
3: addi $a0, $a0, 4
4: bne $a0, $t0, LOOP
5: lw $t1, 0($a0)
6: add $v0, $v0, $t1
7: addi $a0, $a0, 4
8: bne $a0, $t0, LOOP

- Instructions without dependencies can be executed in parallel or out-of-order
- Instructions with dependencies can never be reordered
Limitation of compiler optimizations

- Compiler can only optimize “static instructions”
  - The left-hand side in the table
  - Compiler cannot re-order 2, 5 and 4,5
    - Hardware can do this with branch prediction
- Compiler optimization is constrained by false dependencies due to limited number of registers
  - Instructions 1, 3 do not depend on each other

<table>
<thead>
<tr>
<th>static instructions</th>
<th>dynamic instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOOP: lw $t1, 0($a0)</td>
<td>1: lw $t1, 0($a0)</td>
</tr>
<tr>
<td>add $v0, $v0, $t1</td>
<td>2: add $v0, $v0, $t1</td>
</tr>
<tr>
<td>addi $a0, $a0, 4</td>
<td>3: addi $a0, $a0, 4</td>
</tr>
<tr>
<td>bne $a0, $t0, LOOP</td>
<td>4: bne $a0, $t0, LOOP</td>
</tr>
<tr>
<td>lw $t0, 0($sp)</td>
<td>5: lw $t1, 0($a0)</td>
</tr>
<tr>
<td>lw $t1, 4($sp)</td>
<td>6: add $v0, $v0, $t1</td>
</tr>
<tr>
<td></td>
<td>7: addi $a0, $a0, 4</td>
</tr>
<tr>
<td></td>
<td>8: bne $a0, $t0, LOOP</td>
</tr>
</tbody>
</table>
False dependencies

- They are not “true” dependencies because they don’t have an arrow in data dependency graph
  - WAR (Write After Read): a later instruction overwrites the source of an earlier one
    - 1 and 3, 5 and 7
  - WAW (Write After Write): a later instruction overwrites the output of an earlier one
    - 1 and 5

1:  lw  $t1, 0($a0)
2:  add $v0, $v0, $t1
3:  addi $a0, $a0, 4
4:  bne $a0, $t0, LOOP
5:  lw  $t1, 0($a0)
6:  add $v0, $v0, $t1
7:  addi $a0, $a0, 4
8:  bne $a0, $t0, LOOP
False dependencies

• Consider the following code
  1: lw $t2, 0($a0)
  2: add $t2, $t0, $t2
  3: sub $t8, $t2, $t0
  4: lw $t2, 4($a0)
  5: add $t4, $t8, $t2
  6: add $t8, $t4, $t4
  7: sw $t4, 8($a0)
  8: addi $a0, $a0, 4

which of the following pair is not a “false dependency”

A. 1 and 4  WAW
B. 1 and 8  WAR
C. 5 and 7  True dependency (RAW)
D. 4 and 8  WAR
E. 7 and 8  WAR
Out-of-order processor design
Register renaming

• We can remove false dependencies if we can store each new output in a different register
• Maintains a map between “physical” and “architectural” registers
• Architectural registers: an abstraction of registers visible to compilers and programmers
• Physical registers: the internal registers used for execution
  • Larger number than architectural registers
  • Modern processors have 128 physical registers
Register renaming

Original code

1: lw $t1, 0($a0)
2: add $v0, $v0, $t1
3: addi $a0, $a0, 4
4: bne $a0, $t0, LOOP
5: lw $t1, 0($a0)
6: add $v0, $v0, $t1
7: addi $a0, $a0, 4
8: bne $a0, $t0, LOOP

After renamed

1: lw $p5, 0($p1)
2: add $p6, $p4, $p5
3: addi $p7, $p1, 4
4: bne $p7, $p2, LOOP
5: lw $p8, 0($p7)
6: add $p9, $p6, $p8
7: addi $p10, $p7, 4
8: bne $p10, $p2, LOOP

Register map

<table>
<thead>
<tr>
<th>cycle</th>
<th>$a0</th>
<th>$t0</th>
<th>$t1</th>
<th>$v0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>p1</td>
<td>p2</td>
<td>p3</td>
<td>p4</td>
</tr>
<tr>
<td>1</td>
<td>p1</td>
<td>p2</td>
<td>p5</td>
<td>p4</td>
</tr>
<tr>
<td>2</td>
<td>p1</td>
<td>p2</td>
<td>p5</td>
<td>p6</td>
</tr>
<tr>
<td>3</td>
<td>p7</td>
<td>p2</td>
<td>p5</td>
<td>p6</td>
</tr>
<tr>
<td>4</td>
<td>p7</td>
<td>p2</td>
<td>p5</td>
<td>p6</td>
</tr>
<tr>
<td>5</td>
<td>p7</td>
<td>p2</td>
<td>p8</td>
<td>p6</td>
</tr>
<tr>
<td>6</td>
<td>p7</td>
<td>p2</td>
<td>p8</td>
<td>p9</td>
</tr>
<tr>
<td>7</td>
<td>p10</td>
<td>p2</td>
<td>p8</td>
<td>p9</td>
</tr>
<tr>
<td>8</td>
<td>p10</td>
<td>p2</td>
<td>p8</td>
<td>p9</td>
</tr>
</tbody>
</table>
OOO processor pipeline

- IF stage fetches several instruction in program order
- ID stage decodes instructions and put decoded instructions into “instruction window”
- A new “schedule” stage examines the data dependencies of instructions
  - Send instruction to EXE stage if all source operands are ready
  - The larger the instruction window is, the more ILP we can extract
- But...
  - Logic of instruction window is complex!
  - Keeping the instruction window filled is challenging, because we have branches for every 4-5 instructions.
The instruction window
Scheduling across branches

- Hardware can schedule instruction across branch instructions with the help of branch prediction
- Fetch instructions according to the branch prediction
- Execute instructions across branches
  - Speculative execution: execute an instruction before the processor know if we need to execute or not
  - Execute an instruction all operands are ready (the values of depending physical registers are generated)
  - Store results in “reorder buffer” before the processor knows if the instruction is going to be executed or not.
Reorder buffer

- An instruction will be given an reorder buffer entry number
- A instruction can “retire”/ “commit” only if all its previous instructions finishes.
- If branch mis-predicted, “squash” all instructions with later reorder buffer indexes and clear the occupied physical registers
- We can implement the reorder buffer by extending instruction window or the register map.
Simplified OOO pipeline
Dynamic execution with register naming

- Register renaming, dynamical scheduling with 2-issue pipeline
- Assume that we fetch/decode/renaming/retire 4 instructions into/from instruction window each cycle

After renamed

1: lw $p5, 0($p1)
2: add $p6, $p4, $p5
3: addi $p7, $p1, 4
4: bne $p7, $p2, LOOP
5: lw $p8, 0($p7)
6: add $p9, $p6, $p8
7: addi $p10, $p7, 4
8: bne $p10, $p2, LOOP
Dynamic execution with register naming

- Register renaming, dynamical scheduling with 2-issue pipeline
- Assume that we fetch/decode/renaming/retire 4 instructions into/from instruction window each cycle

1: lw $p5, 0($p1)
2: add $p6, $p4, $p5
3: addi $p7, $p1, 4
4: bne $p7, $p2, LOOP
5: lw $p8, 0($p7)
6: add $p9, $p6, $p8
7: addi $p10, $p7, 4
8: bne $p10, $p2, LOOP
Example: Alpha 21264

- **Fetch**
  - Branch predictor
  - Instruction prefetcher
  - Instruction Cache

- **Slot Rename**
  - Register renaming logic

- **Issue**
  - Issue queue

- **Register Read**
  - Register file

- **Execute**
  - Execute units

- **Memory**
  - Data cache
Example: intel Nehalem

Intel Nehalem Execution Engine

28-entry Instruction Decode Queue

128-entry Reorder Buffer

36-entry Unified Reservation Station

Port 0

Port 1

Port 2

Port 3

Port 4

Port 5

Integer ALU/Shift

Integer ALU/LEA

Load

Store Address

Store Data

Integer ALU/Shift

FP Mul

FP Add

Vector Int Mul

Vector Int ALU

Vector Logicals

Vector Logicals

Vector Logicals

Vector Logicals

Vector Shifts

Divide

Vector Int ALU

Vector Logicals

Branch
Problems with OOO+Superscalar

• The modern OOO processors have 3-5 issue widths
• Keeping instruction window filled is hard
  • Branches are every 4-5 instructions.
  • If the instruction window is 32 instructions the processor has to predict 6-8 consecutive branches correctly to keep IW full.

• Hardware complexity & Area Efficiency
  • Multi-ported Register File ~ IW4
  • IQ size ~ IW4
  • Bypass logic ~ IW4
  • Wiring delay

• The ILP within an application is low
  • Usually 1-2 per thread
  • ILP is even lower is data depends on memory operations (if cache misses) or long latency operations
The instruction window

Decoded Instruction data

alu_out_dst_0

alu_out_dst_1

alu_out_value_0

alu_out_value_1

rs_value

valid

rt_value

valid

Ready

rt_value

rs_value

opcode
Q & A