Modern processor design

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

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 have a perfect branch predictor

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

5 cycles per loop in average: IPC = 0.8
Instruction level parallelism

• We have used pipeline to shrink the cycle time as short as possible
• Pipeline processors increase 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?
Outline

• SuperScalar
• Dynamic scheduling/Out-of-order execution
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 = 1/n for an n-issue SS processor in the best case.

```c
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
```

2 cycle per loop with perfect branch prediction: CPI = 0.5!
Pipeline takes 4 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
```

4 cycles if the processor predicts branch perfectly
IPC = 1, not very impressive...
Reordering using compiler

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

2 cycles if the processor predicts branch perfectly
Limitations of compiler

- Static instructions: instructions in compiled code
- Dynamic instructions: the real instruction stream when running the program
- Compiler can only see/optimize **static instructions**
  - The left-hand side in the table
  - Compiler cannot re-order 3, 5 and 4,5
- Compiler optimization is constrained by **false dependencies** due to limited number of registers
  - Instructions 1 and 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>
Out-of-order processor design
Designing the front-end of an out-of-order processor

- Reorder instructions on **dynamic instructions**
  - Fetches multiple instructions at the same time
  - Fetches instructions across the branch: with the help of branch prediction
- Schedule the instructions into execution if there is no pending data dependencies
The instruction window

Schedule

Execute
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
False dependencies

• We are still limited by **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
If we can transform the 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
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>
Simplified OOO pipeline

Instruction Fetch → Instruction Decode → Register renaming logic → Schedule → Execution Units → Data Memory → Write Back

Branch predictor
Scheduling across branches

• Hardware can schedule instruction across branch instructions with the help of branch prediction
  • Fetch instructions according to the branch prediction
  • However, branch predictor can never be perfect
• 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, “flush” 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

Instruction Fetch → Instruction Decode → Register renaming logic → Schedule → Execution Units → Data Memory → Reorder Buffer/Commit

Branch predictor
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
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  slot  rename  issue  register  read  execute  memory

Branch predictor

Instruction prefetcher

Instruction Cache

Register renaming logic

Issue queue

Register file

Execute units

Data cache
AMD K10 architecture
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 Int ALU

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.
- 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
- Hardware complexity & Area Efficiency
  - Multi-ported Register File $\sim (IW)^4$
  - IQ size $\sim (IW)^4$
  - Data forwarding logic $\sim (IW)^4$
  - Wiring delay