Modern processor design

Hung-Wei Tseng
Let’s start from this code

```assembly
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 \texttt{0x10000000} and $t0 is \texttt{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: CPI = 1.25
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)           IF  ID  EXE  MEM  WB
addi $a0, $a0, 4          IF  ID  EXE  MEM  WB
add  $v0, $v0, $t1         IF  ID  EXE  MEM  WB
bne  $a0, $t0, LOOP        IF  ID  EXE  MEM  WB
lw   $t1, 0($a0)           IF  ID  EXE  MEM  WB
addi $a0, $a0, 4          IF  ID  EXE  MEM  WB
add  $v0, $v0, $t1         IF  ID  EXE  MEM  WB
bne  $a0, $t0, LOOP        IF  ID  EXE  MEM  WB
```

4 cycles per loop in average: CPI = 1
Instruction level parallelism

- The ability of execution multiple instructions at the same time
- We have used pipeline to shrink the cycle time as short as possible
- Pipeline processors increase the throughput by improving instruction level parallelism (ILP)
- 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.

```
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
Running compiler optimized code

- 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
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
```

3 cycles if the processor predicts branch perfectly, CPI = 0.75

Can further improve performance if we can reorder this...
Limitations of compiler optimizations

- Compiler can only see/optimize **static instructions**, instructions in the compiled binary
- Compiler cannot optimize **dynamic instructions**, the real instruction sequence when executing the program
  - Compiler cannot re-order 3, 5 or 4,5
  - Compiler cannot predict cache misses
- Compiler optimization is constrained by **false dependencies** due to limited number of registers
  - Instructions `lw $t1, 0($a0)` and `addi $a0, $a0, 4` do not depend on each other
- Compiler cannot optimize for architecture dependent features
  - The code optimization in the previous example works for single pipeline, but not for superscalar

<table>
<thead>
<tr>
<th>Static instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOOP: <code>lw $t1, 0($a0)</code></td>
</tr>
<tr>
<td><code>addi $a0, $a0, 4</code></td>
</tr>
<tr>
<td><code>add $v0, $v0, $t1</code></td>
</tr>
<tr>
<td><code>bne $a0, $t0, LOOP</code></td>
</tr>
<tr>
<td><code>lw $t0, 0($sp)</code></td>
</tr>
<tr>
<td><code>lw $t1, 4($sp)</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dynamic instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: <code>lw $t1, 0($a0)</code></td>
</tr>
<tr>
<td>2: <code>addi $a0, $a0, 4</code></td>
</tr>
<tr>
<td>3: <code>add $v0, $v0, $t1</code></td>
</tr>
<tr>
<td>4: <code>bne $a0, $t0, LOOP</code></td>
</tr>
<tr>
<td>5: <code>lw $t1, 0($a0)</code></td>
</tr>
<tr>
<td>6: <code>addi $a0, $a0, 4</code></td>
</tr>
<tr>
<td>7: <code>add $v0, $v0, $t1</code></td>
</tr>
<tr>
<td>8: <code>bne $a0, $t0, LOOP</code></td>
</tr>
</tbody>
</table>
Out-of-order processor design
Designing an out-of-order processor

- The goal is to “reorder/optimize instructions using dynamic instructions”
  - Needs to fetch multiple instructions at the same time so that we have more instructions to schedule
  - Needs the help of branch prediction to fetches instructions across the branch
- The hardware can schedule the execution of these fetched instructions
Scheduling instructions: based on data dependencies

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

- In theory, 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 2, 3 and 5, 5 and 6
  - **WAW (Write After Write):** a later instruction overwrites the output of an earlier one
    - 1 and 5

```
1: lw   $t1, 0($a0)
2: addi $a0, $a0, 4
3: add  $v0, $v0, $t1
4: bne  $a0, $t0, LOOP
5: lw   $t1, 0($a0)
6: addi $a0, $a0, 4
7: add  $v0, $v0, $t1
8: bne  $a0, $t0, LOOP
```
If we can transform the code ...

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

1: lw   $t1, 0($a0)
2: addi $a1, $a0, 4
3: add  $v1, $v0, $t1
4: bne  $a1, $t0, LOOP
5: lw   $t2, 0($a1)
6: addi $a2, $a1, 4
7: add  $v2, $v1, $t2
8: bne  $a2, $t0, LOOP

- We can get rid of the problem if each new output can use a different register!
- Compiler cannot do this because compiler cannot know if the second loop will be executed or not!
Register renaming

• We can remove false dependencies if we can store each new output in a different register

• Architectural registers: an abstraction of registers visible to compilers and programmers
  • Like MIPS $0 -- $31

• Physical registers: the internal registers used for execution
  • Larger number than architectural registers
  • Modern processors have 128 physical registers
  • Invisible to programmers and compilers

• Maintains a mapping table between “physical” and “architectural” registers
Register renaming

Original code

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

After renamed

1: lw $p5 , 0($p1)
2: addi $p6 , $p1, 4
3: add $p7 , $p4, $p5
4: bne $p6 , $p2, LOOP
5: lw $p8 , 0($p6)
6: addi $p9 , $p6, 4
7: add $p10, $p7, $p8
8: bne $p9 , $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>p6</td>
<td>p2</td>
<td>p5</td>
<td>p4</td>
</tr>
<tr>
<td>3</td>
<td>p6</td>
<td>p2</td>
<td>p5</td>
<td>p7</td>
</tr>
<tr>
<td>4</td>
<td>p6</td>
<td>p2</td>
<td>p5</td>
<td>p7</td>
</tr>
<tr>
<td>5</td>
<td>p6</td>
<td>p2</td>
<td>p8</td>
<td>p7</td>
</tr>
<tr>
<td>6</td>
<td>p9</td>
<td>p2</td>
<td>p8</td>
<td>p7</td>
</tr>
<tr>
<td>7</td>
<td>p9</td>
<td>p2</td>
<td>p8</td>
<td>p10</td>
</tr>
<tr>
<td>8</td>
<td>p9</td>
<td>p2</td>
<td>p8</td>
<td>p10</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.
The instruction window

Schedule

Execute
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
- Assume load needs 2 cycles to execute (one cycle address calculation and one cycle memory access)

After renamed

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

Available in cycle #1:
- cycle #1
- cycle #2

Available in cycle #2:
- cycle #3
- cycle #4

Cannot issue because the issue width is only 2
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: addi $p6, $p1, 4
3: add $p7, $p4, $p5
4: bne $p6, $p2, LOOP
5: lw $p8, 0($p6)
6: addi $p9, $p6, 4
7: add $p10, $p7, $p8
8: bne $p9, $p2, LOOP

Execute/issue 2 instructions every cycle, CPI = 0.5
Example: Alpha 21264
AMD K10 architecture

- ITLB: 48-entry
- Level 1 Instruction Cache: 64 KByte
- 256 Bit + 15 Bit152 Predecode, Branch, Parity
- Level 2 Cache: 512K, 16-way exclusive
- L2 ECC
- L2 Tags
- L2 Tag ECC
- Decode
  - DirectPath
  - VectorPath
- Instruction Control Unit (72-entry)
- 8-entry Scheduler
- 8-entry Scheduler
- 8-entry Scheduler
- 36-entry Scheduler
- Load / Store Queue (44-entry)
- Data TLB: 48-entry
- Level 1 Data Cache: 64 KByte, 2-way ECC

- (zusammen für alle vier Core(s))

- Red: Difference between K8 and K10 Architecture
  (Die Änderungen zwischen der K8- und K10-Architektur sind rot markiert)
Example: intel Nehalem

Intel Nehalem Execution Engine

28-entry Instruction Decode Queue → 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 ALU

Vector Int Mul → Vector Int ALU

Vector Logicals → Vector Logicals

Vector Shifts

Divide → 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