Modern processor architectures (I)

Hung-Wei Tseng
Let’s revisit the current pipeline

If the current value of
$a0$ is $0x10000000$ and
$t0$ is $0x10001000$, what are the
dynamic instructions that the
processor will execute?
• Draw the pipeline execution diagram
  • assume that we have full data forwarding path
  • assume that we have a perfect branch predictor

```assembly
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
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 without affecting correctness?

A. 1 and 3
B. 2 and 3
C. 2 and 4
D. 3 and 4
E. No room for optimizations
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)
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
```

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

- The ability of execution multiple instructions at the same cycle
- We have used pipeline to shrink the cycle time
- Pipeline processors increase the throughput by improving instruction level parallelism (ILP)
- With data forwarding, branch prediction and caches, we still can only achieve $\text{CPI} = 1$ in the best case.

Can we further improve ILP to achieve CPI < 1?
Outline

- SuperScalar
- Dynamic instruction scheduling + Out-of-order execution
SuperScalar
SuperScalar

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 cycles per iteration if the processor predicts branch perfectly, CPI 2/4 = 0.5!
However, most of time, your program looks like this …

These instructions are not born equal; the popularity of the few dominates the many. For example, Figure 2.45 shows the popularity of each class of instructions for SPEC CPU2006. The varying popularity of instructions plays an important role in the chapters about datapath, control, and pipelining.

<table>
<thead>
<tr>
<th>Instruction class</th>
<th>MIPS examples</th>
<th>HLL correspondence</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Integer</td>
</tr>
<tr>
<td>Arithmetic</td>
<td>add, sub, addi</td>
<td>Operations in assignment statement s</td>
<td>16%</td>
</tr>
<tr>
<td>Data transfer</td>
<td>lw, sw, lb, lbu, lh, lhu, sb, lui</td>
<td>References to data structures, such as arrays</td>
<td>35%</td>
</tr>
<tr>
<td>Logical</td>
<td>and, or, nor, andi, ori, sll, srl</td>
<td>Operations in assignment statement s</td>
<td>12%</td>
</tr>
<tr>
<td>Conditional branch</td>
<td>beq, bne, slt, slti, sltiu</td>
<td>If statements and loops</td>
<td>34%</td>
</tr>
<tr>
<td>Jump</td>
<td>j, jr, jal</td>
<td>Procedure calls, returns, and case/switch statements</td>
<td>2%</td>
</tr>
</tbody>
</table>

**FIGURE 2.45** MIPS instruction classes, examples, correspondence to high-level program language constructs, and percentage of MIPS instructions executed by category for the average integer and floating point SPEC CPU2006 benchmarks. Figure 3.26 in Chapter 3 shows average percentage of the individual MIPS instructions executed.
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

Can further improve performance if we can reorder this...

3 cycles if the processor predicts branch perfectly, CPI = 0.75
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 (even worse for x86).
  - Instructions `lw $t1, 0($a0)` and `addi $a0, $a0, 4` do not depend on each other.
- Compiler optimizations do not work for all architectures.
  - The code optimization in the previous example works for single pipeline, but not for superscalar.

### Static instructions

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

### Dynamic instructions

```
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
```
Simply superscalar + compiler optimization is not enough
Limitations of compiler optimizations

- Compiler cannot optimize dynamic instructions, the real instruction sequence when executing the program
  - Cannot reorder across branches most of time
  - Cannot handle cases like cache misses
- Compiler optimizations do not work for all architectures
  - The code optimization in the previous example works for single pipeline, but not for superscalar
  - The code optimization may not work well if we change the pipeline design
- Compiler optimization is constrained by the ISA that processor exposes to the software
  - Can only optimize code use limited number of registers
  - False dependencies due to limited number of registers (even worse for x86)
Dynamic/OoO instruction scheduling
Basic idea — when can we execute an instruction?

- Whenever the instruction is decoded — put decoded instruction somewhere
- Whenever the inputs are ready — all data dependencies are resolved
- Whenever the target functional unit is available
The goal is to “reorder/optimize instructions using dynamic instructions”

- Needs to fetch more instructions than the number of functional units at the same time so that we have more instructions to schedule
- Needs to store decoded instructions that are pending somewhere
- Needs the help of branch prediction to fetches instructions across the branch
- The hardware can schedule the execution of these fetched instructions — based on the availability of inputs and functional units
The instruction queue & schedule

Schedule

Execute
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

• 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 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.
Speculative execution

- Exceptions may occur anytime
  - A later instruction cannot write back its own result otherwise the architectural states won’t be correct
- 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 supporting speculation

• 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 queue or the register map.
Dynamic execution with register naming

- Register renaming with unlimited physical registers, 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)
Dynamic execution with register naming

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

Execute these instructions out-of-order

Execute/issue 2 instructions per cycle, CPI = 0.5
Problems with OOO+Superscalar

- The modern OOO processors have 3-6 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
- Demo
Example: Alpha 21264
Simplified OOO pipeline

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

Branch predictor
Alpha 21264

fetch → slot → rename → issue → register → execute → memory

- Branch predictor
- Instruction prefetcher
- Instruction Cache
- Register renaming logic
- Issue queue
- Register file
- Execute units
- Data cache
Pentium 4
Intel SkyLake

The Skylake microarchitecture builds on the successes of the Haswell and Broadwell microarchitectures. The basic pipeline functionality of the Skylake microarchitecture is depicted in Figure 2-1.

The Skylake microarchitecture offers the following enhancements:

- Larger internal buffers to enable deeper OOO execution and higher cache bandwidth.
- Improved front end throughput.
- Improved branch predictor.
- Improved divider throughput and latency.
- Lower power consumption.
- Improved SMT performance with Hyper-Threading Technology.
- Balanced floating-point ADD, MUL, FMA throughput and latency.

The microarchitecture supports flexible integration of multiple processor cores with a shared uncore subsystem consisting of a number of components including a ring interconnect to multiple slices of L3 (an off-die L4 is optional), processor graphics, integrated memory controller, interconnect fabrics, etc. A four-core configuration can be supported similar to the arrangement shown in Figure 2-3.

Figure 2-1. CPU Core Pipeline Functionality of the Skylake Microarchitecture
Problems with OOO+Superscalar

- The modern OOO processors have 3-6 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
- Demo