Modern processor design

Hung-Wei Tseng
Let’s revisit the current pipeline

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 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
Compiler optimization: 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 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)  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 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 CPI = 1 in the best case.
• Can we further improve ILP to achieve CPI < 1?
Announcement

- Reading quiz due tomorrow
- Hung-Wei’s office hour this week
  - Thursday: 10:30a-11:30a
  - Friday: TBD
- Grades calculation
  - Dropping the lowest homework
  - Dropping 3 lowest quizzes
  - Already reflected in your weighted total
Outline

- SuperScalar
- Dynamic scheduling/Out-of-order execution
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.

```assembly
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

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.
Simply superscalar + compiler optimization is not enough
Dynamic out-of-order execution
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
The instruction window

Schedule

Execute

insts

Arbitration

ALU0

ALU1
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
Scheduling across the branch

- Consider the following 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

- Which of the following pair can we reorder without affecting the correctness if the branch prediction is perfect?
  A. 1 and 2
  B. 3 and 5
  C. 3 and 6
  D. 4 and 5
  E. 4 and 6
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)  # 1
2: addi $a0, $a0, 4  # 2
3: add  $v0, $v0, $t1 # 3
4: bne  $a0, $t0, LOOP # 4
5: lw   $t1, 0($a0)  # 5
6: addi $a0, $a0, 4  # 6
7: add  $v0, $v0, $t1 # 7
8: bne  $a0, $t0, LOOP # 8
```
False dependencies

- Consider the following dynamic instructions:
  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
B. 1 and 8
C. 5 and 7
D. 4 and 8
E. 7 and 8
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>
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
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)

```
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
```
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

```assembly
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 these instructions out-of-order

Execute/issue 2 instructions per cycle, CPI = 0.5
Dynamic execution with register naming

- Consider the following dynamic instructions:
  1: `lw $t1, 0($a0)`
  2: `lw $a0, 4($a0)`
  3: `add $v0, $v0, $t1`
  4: `bne $a0, $zero, LOOP`  
  5: `lw $t1, 0($a0)`
  6: `lw $t2, 4($a0)`
  7: `add $v0, $v0, $t1`
  8: `bne $t2, $zero, LOOP`

Assume a superscalar processor with unlimited issue width & physical registers that can fetch up to 4 instructions per cycle, 2 cycles to execute a memory instruction. How many cycles it takes to issue all instructions?

A. 1  
B. 2  
C. 3  
D. 4  
E. 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