Out-of-order Superscalar

740
October 31, 2014

So Far …

CPU Time = CPU clock cycles \times clock cycle time
= \frac{\text{instructions}}{\text{program}} \times \frac{\text{cycles}}{\text{instruction}} \times \frac{\text{seconds}}{\text{cycle}}

- Increase clock frequency
- Decrease CPI
- Pipeline:
  - divide CPI by # of stages + bubbles
  - add bypass
- Superscalar
  - divide CPI by issue width + bubbles from
    » data dependencies
    » control dependencies

What next?

• What slows us down here?

// for (i=0; i<N; i++) c[i] = a[i]*b[i]+i*2;
loop:
  ld r1, (r2+r9*8)
  ld r3, (r4+r9*8)
  mult r5, r3, r1
  mult r3, r9, 2
  add r5, r5, r3
  st r5, (r6+r9*8)
  add r9, r9, 1
  cmp r9, r10
  bnz loop

What next?

• What slows us down here?

// for (i=0; i<N; i++) c[i] = a[i]*b[i]+i*2;
loop:
  ld r1, (r2+r9*8)
  ld r3, (r4+r9*8)
  mult r5, r3, r1
  mult r3, r9, 2
  add r5, r5, r3
  st r5, (r6+r9*8)
  add r9, r9, 1
  cmp r9, r10
  bnz loop
How can we improve this?

• Assume:
  • 2-issue superscalar
  • 1 LD/ST unit (2 cycles)
  • 1 mult (2 cycles)
  • 1 add (1 cycle)

```
loop:  ld  r1, (r2+r9*8)
      ld  r3, (r4+r9*8)
      mult r5, r3, r1
      mult r3, r9, 2
      add r5, r5, r3
      st  r5, (r6+r9*8)
      add r9, r9, 1
      cmp r9, r10
      st  r5, (r6+r9*8)
      add r9, r9, 1
      cmp r9, r10
      bnez loop
```

What next?

• What slows us down here?

```
loop:  ld  r1, (r2+r9*8)
      ld  r3, (r4+r9*8)
      mult r5, r3, r1
      mult r3, r9, 2
      add r5, r5, r3
      st  r5, (r6+r9*8)
      add r9, r9, 1
      cmp r9, r10
      bnez loop
```

• Data Dependencies
  • RAW: Read after write. (true dependence)
  • WAR: write after read (false dependence)
  • WAW: write after write (false dependence)
Eliminating WAR (and WAW)

• What slows us down here?

  loop: ld r1, (r2+r9*8)
  ld r3, (r4+r9*8)
  mult r5, r3, r1
  mult r11, r9, 2
  add r5, r5, r11
  st r5, (r6+r9*8)
  add r12, r9, 1
  cmp r12, r10
  mov r9, r12
  bnz loop

• Data Dependencies

  • RAW: Read after write. (true dependence)
  • WAR: write after read (false dependence)
  • WAW: write after write (false dependence)

Reschedule

loop: ld r1, (r2+r9*8)
  mult r11, r9, 2
  ld r3, (r4+r9*8)
  mult r5, r3, r1
  mult r11, r9, 2
  add r5, r5, r11
  st r5, (r6+r9*8)
  add r12, r9, 1
  cmp r12, r10
  mov r9, r12
  bnz loop

Can we do better?

loop: ld r1, (r2+r9*8)
  mult r11, r9, 2
  ld r3, (r4+r9*8)
  mult r5, r3, r1
  ld r3, (r4+r9*8)
  mult r5, r3, r1
  add r5, r5, r11
  st r5, (r6+r9*8)
  add r12, r9, 1
  cmp r12, r10
  mov r9, r12
  bnz loop

7 cycles

More Rescheduling

loop: ld r1, (r2+r9*8)
  mult r11, r9, 2
  ld r3, (r4+r9*8)
  add r12, r9, 1
  mult r5, r3, r1
  cmp r12, r10
  mov r9, r12
  bnz loop

loop: ld r1, (r2+r9*8)
  mult r11, r9, 2
  ld r3, (r4+r9*8)
  add r12, r9, 1
  cmp r12, r10
  mov r9, r12
  bnz loop

7 cycles

10 cycles
What Is Holding Us Back?

• In order issue
• Not enough Function units
• Function units too slow
• Not enough register names
• Control dependence
• Static scheduling

Out-of-Order Superscalar

• In-order fetch & decode
• Out-of-order
  • issue
  • register-read
  • execute
  • memory
  • write-back
• In-order commit

Out-of-Order Pipeline
Out-of-Order Execution

• Also called “Dynamic scheduling”
  • Done by the hardware on-the-fly during execution

• Looks at a “window” of instructions waiting to execute
  • Each cycle, picks the next ready instruction(s)

Two steps to enable out-of-order execution:
  Step #1: Register renaming – to avoid “false” dependencies
  Step #2: Dynamically schedule – to enforce “true” dependencies

Key to understanding out-of-order execution:
  • Data dependencies

Dependence types

• RAW (Read After Write) = “true dependence” (true)
  \[ \text{mul } r0 \times r1 \rightarrow r2 \]
  \[ \text{add } r2 \rightarrow r3 \rightarrow r4 \]

• WAW (Write After Write) = “output dependence” (false)
  \[ \text{mul } r0 \times r1 \rightarrow r2 \]
  \[ \text{add } r1 + r3 \rightarrow r2 \]

• WAR (Write After Read) = “anti-dependence” (false)
  \[ \text{mul } r0 \times r1 \rightarrow r2 \]
  \[ \text{add } r3 + r4 \rightarrow r1 \]

• WAW & WAR are “false”. Can be totally eliminated by “renaming”

Step #1: Register Renaming

• To eliminate register conflicts/hazards
  • “Architected” vs “Physical” registers – level of indirection
    • Names: r1, r2, r3
    • Locations: p1, p2, p3, p4, p5, p6, p7
    • Original mapping: r1 \rightarrow p1, r2 \rightarrow p2, r3 \rightarrow p3, p4 \rightarrow p7 are “available”

<table>
<thead>
<tr>
<th>MapTable</th>
<th>FreeList</th>
<th>Original insns</th>
<th>Renamed insns</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1 p2 p3</td>
<td>p4, p5, p6, p7</td>
<td>add r2, r3 \rightarrow r1</td>
<td>add p2, p3 \rightarrow p4</td>
</tr>
<tr>
<td>p4 p2 p3</td>
<td>p5, p6, p7</td>
<td>sub r2, r1 \rightarrow r3</td>
<td>sub p2, p3 \rightarrow p5</td>
</tr>
<tr>
<td>p4 p2 p5</td>
<td>p6, p7</td>
<td>mul r2, r3 \rightarrow r1</td>
<td>mul p2, p6 \rightarrow p5</td>
</tr>
<tr>
<td>p4 p2 p6</td>
<td>p7</td>
<td>div r1 \rightarrow r1</td>
<td>div p4, p7</td>
</tr>
</tbody>
</table>

• Renaming – conceptually write each register once
  + Removes false dependences
  + Leaves true dependences intact!
• When to reuse a physical register? After overwriting insn done

Register Renaming Algorithm

• Two key data structures:
  • maptable[architectural_reg] \rightarrow physical_reg
  • Free list: allocate (new) & free registers (implemented as a queue)

• Algorithm: at “decode” stage for each instruction:
  scanf.phys_input1 = maptable[scanf.arch_input1]
  scanf.phys_input2 = maptable[scanf.arch_input2]
  scanf.old_phys_output = maptable[scanf.arch_output]
  insn.old_phys_output = maptable[insn.arch_output]
  new_reg = new_phys_reg()
  maptable[insn.arch_output] = new_reg
  insn.phys_output = new_reg

• At “commit”
  • Once all older instructions have committed, free register
    free_phys_reg(insn.old_phys_output)
Out-of-order Pipeline

- Fetch
- Decode
- Rename
- Dispatch
- Issue
- Reg-read
- Execute
- Writeback
- Commit

Have unique register names
Now put into out-of-order execution structures

Step #2: Dynamic Scheduling

- Instructions fetch/decoded/renamed into Instruction Buffer
  - Also called “instruction window” or “instruction scheduler”
- Instructions (conceptually) check ready bits every cycle
  - Execute oldest “ready” instruction, set output as “ready”

Dynamic Scheduling/Issue Algorithm

- Data structures:
  - Ready table[phys_reg] ➔ yes/no (part of “issue queue”)

- Algorithm at “issue” stage (prior to read registers):
  foreach instruction:
    if table[insn.phys_input1] == ready &&
      table[insn.phys_input2] == ready
    then
      insn is “ready”
      select the oldest “ready” instruction
      table[insn.phys_output] = ready

- Multiple-cycle instructions? (such as loads)
  - For an insn with latency of N, set “ready” bit N-1 cycles in future

Cycle 0

<table>
<thead>
<tr>
<th>Arch</th>
<th>Phys</th>
<th>Reg</th>
<th>Value</th>
<th>Ready</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td>p1</td>
<td>p1</td>
<td>n</td>
<td></td>
</tr>
<tr>
<td>r2</td>
<td>p2</td>
<td>p2</td>
<td>a</td>
<td>Y</td>
</tr>
<tr>
<td>r3</td>
<td>p3</td>
<td>p3</td>
<td>n</td>
<td></td>
</tr>
<tr>
<td>r4</td>
<td>p4</td>
<td>p4</td>
<td>b</td>
<td>Y</td>
</tr>
<tr>
<td>r5</td>
<td>p5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r6</td>
<td>p6</td>
<td>p6</td>
<td>c</td>
<td>Y</td>
</tr>
<tr>
<td>r7</td>
<td>p7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r8</td>
<td>p8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r9</td>
<td>p9</td>
<td>p9</td>
<td>i</td>
<td>Y</td>
</tr>
<tr>
<td>r10</td>
<td>p10</td>
<td>p10</td>
<td>n</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td></td>
<td>p11</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>p12</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>p13</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>p14</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

loop: ld r1, (r2+r9*8)
      ld r3, (r4+r9*8)
      mult r5, r3, r1
      mult r5, r9, 2
      add r5, r5, r3
      st r5, (r6+r9*8)
      add r9, r9, 1
      cmp r9, r10
      bnz loop

ld r1, (p2+p9*8)
ld r3, (p4+p9*8)
Cycle 1

loop: ld r1, (r2+r9*8)
    ld r3, (r4+r9*8)
    mult r5, r3, r1
    mult r3, r9, 2
    add r5, r5, r3
    st r5, (r6+r9*8)
    add r9, r9, 1
    cmp r9, r10
    bnz loop

ld p1, (p2+p9*8)
    ld p3, (p4+p9*8)
    mult p5, p3, p1
    mult p11, p9, 2
    add     p12, p5, p11
    st p12, (p6+p9*8)

Cycle 2

loop: ld r1, (r2+r9*8)
    ld r3, (r4+r9*8)
    mult r5, r3, r1
    mult r3, r9, 2
    add r5, r5, r3
    st r5, (r6+r9*8)
    add r9, r9, 1
    cmp r9, r10
    bnz loop

ld p1, (p2+p9*8)
    ld p3, (p4+p9*8)
    mult p5, p3, p1
    mult p11, p9, 2
    add     p12, p5, p11
    st p12, (p6+p9*8)

Cycle 3

loop: ld r1, (r2+r9*8)
    ld r3, (r4+r9*8)
    mult r5, r3, r1
    mult r3, r9, 2
    add r5, r5, r3
    st r5, (r6+r9*8)
    add r9, r9, 1
    cmp r9, r10
    bnz loop

ld p1, (p2+p9*8)
    ld p3, (p4+p9*8)
    mult p5, p3, p1
    mult p11, p9, 2
    add     p12, p5, p11
    st p12, (p6+p9*8)

Cycle 4

loop: ld r1, (r2+r9*8)
    ld r3, (r4+r9*8)
    mult r5, r3, r1
    mult r3, r9, 2
    add r5, r5, r3
    st r5, (r6+r9*8)
    add r9, r9, 1
    cmp r9, r10
    bnz loop

ld p1, (p2+p9*8)
    ld p3, (p4+p9*8)
    mult p5, p3, p1
    mult p11, p9, 2
    add     p12, p5, p11
    st p12, (p6+p9*8)

ld p1, (p2+p9*8)
    ld p3, (p4+p9*8)
    mult p5, p3, p1
    mult p11, p9, 2
    add     p12, p5, p11
    st p12, (p6+p9*8)

ld p13, p9, 1
    cmp p13, p10

Register Renaming Algorithm (Simplified)

- Two key data structures:
  - maptable[architectural_reg] → physical_reg
  - Free list: allocate (new) & free registers (implemented as a queue)

- Algorithm: at “decode” stage for each instruction:

  insn.phys_input1 = maptable[insn.arch_input1]
  insn.phys_input2 = maptable[insn.arch_input2]

  new_reg = new_phys_reg()
  maptable[insn.arch_output] = new_reg
  insn.phys_output = new_reg
Renaming example

xor r1 ^ r2 \rightarrow r3
add r3 + r4 \rightarrow r4
sub r5 - r2 \rightarrow r3
addi r3 + 1 \rightarrow r1

<table>
<thead>
<tr>
<th>r1</th>
<th>p1</th>
</tr>
</thead>
<tbody>
<tr>
<td>r2</td>
<td>p2</td>
</tr>
<tr>
<td>r3</td>
<td>p3</td>
</tr>
<tr>
<td>r4</td>
<td>p4</td>
</tr>
<tr>
<td>r5</td>
<td>p5</td>
</tr>
</tbody>
</table>

Map table

<table>
<thead>
<tr>
<th>p6</th>
</tr>
</thead>
<tbody>
<tr>
<td>p7</td>
</tr>
<tr>
<td>p8</td>
</tr>
<tr>
<td>p9</td>
</tr>
</tbody>
</table>

Free-list

xor r1 ^ r2 \rightarrow r3
add r3 + r4 \rightarrow r4
sub r5 - r2 \rightarrow r3
addi r3 + 1 \rightarrow r1

<table>
<thead>
<tr>
<th>r1</th>
<th>p1</th>
</tr>
</thead>
<tbody>
<tr>
<td>r2</td>
<td>p2</td>
</tr>
<tr>
<td>r3</td>
<td>p3</td>
</tr>
<tr>
<td>r4</td>
<td>p4</td>
</tr>
<tr>
<td>r5</td>
<td>p5</td>
</tr>
</tbody>
</table>

Map table

<table>
<thead>
<tr>
<th>p6</th>
</tr>
</thead>
<tbody>
<tr>
<td>p7</td>
</tr>
<tr>
<td>p8</td>
</tr>
<tr>
<td>p9</td>
</tr>
</tbody>
</table>

Free-list

Renaming example

xor r1 ^ r2 \rightarrow r3
add r3 + r4 \rightarrow r4
sub r5 - r2 \rightarrow r3
addi r3 + 1 \rightarrow r1

<table>
<thead>
<tr>
<th>r1</th>
<th>p1</th>
</tr>
</thead>
<tbody>
<tr>
<td>r2</td>
<td>p2</td>
</tr>
<tr>
<td>r3</td>
<td>p3</td>
</tr>
<tr>
<td>r4</td>
<td>p4</td>
</tr>
<tr>
<td>r5</td>
<td>p5</td>
</tr>
</tbody>
</table>

Map table

<table>
<thead>
<tr>
<th>p6</th>
</tr>
</thead>
<tbody>
<tr>
<td>p7</td>
</tr>
<tr>
<td>p8</td>
</tr>
<tr>
<td>p9</td>
</tr>
</tbody>
</table>

Free-list

xor p1 ^ p2 \rightarrow p6

<table>
<thead>
<tr>
<th>r1</th>
<th>p1</th>
</tr>
</thead>
<tbody>
<tr>
<td>p6</td>
<td></td>
</tr>
</tbody>
</table>

Map table

Renaming example

xor r1 ^ r2 \rightarrow r3
add r3 + r4 \rightarrow r4
sub r5 - r2 \rightarrow r3
addi r3 + 1 \rightarrow r1

<table>
<thead>
<tr>
<th>r1</th>
<th>p1</th>
</tr>
</thead>
<tbody>
<tr>
<td>r2</td>
<td>p2</td>
</tr>
<tr>
<td>r3</td>
<td>p3</td>
</tr>
<tr>
<td>r4</td>
<td>p4</td>
</tr>
<tr>
<td>r5</td>
<td>p5</td>
</tr>
</tbody>
</table>

Map table

<table>
<thead>
<tr>
<th>p6</th>
</tr>
</thead>
<tbody>
<tr>
<td>p7</td>
</tr>
<tr>
<td>p8</td>
</tr>
<tr>
<td>p9</td>
</tr>
</tbody>
</table>

Free-list

xor p1 ^ p2 \rightarrow p6

<table>
<thead>
<tr>
<th>r1</th>
<th>p1</th>
</tr>
</thead>
<tbody>
<tr>
<td>p6</td>
<td></td>
</tr>
</tbody>
</table>

Map table

Renaming example

xor r1 ^ r2 \rightarrow r3
add r3 + r4 \rightarrow r4
sub r5 - r2 \rightarrow r3
addi r3 + 1 \rightarrow r1

<table>
<thead>
<tr>
<th>r1</th>
<th>p1</th>
</tr>
</thead>
<tbody>
<tr>
<td>r2</td>
<td>p2</td>
</tr>
<tr>
<td>r3</td>
<td>p3</td>
</tr>
<tr>
<td>r4</td>
<td>p4</td>
</tr>
<tr>
<td>r5</td>
<td>p5</td>
</tr>
</tbody>
</table>

Map table

<table>
<thead>
<tr>
<th>p7</th>
</tr>
</thead>
<tbody>
<tr>
<td>p8</td>
</tr>
<tr>
<td>p9</td>
</tr>
</tbody>
</table>

Free-list

xor p1 ^ p2 \rightarrow p6

<table>
<thead>
<tr>
<th>r1</th>
<th>p1</th>
</tr>
</thead>
<tbody>
<tr>
<td>p7</td>
<td></td>
</tr>
</tbody>
</table>

Map table

Renaming example

xor r1 ^ r2 \rightarrow r3
add r3 + r4 \rightarrow r4
sub r5 - r2 \rightarrow r3
addi r3 + 1 \rightarrow r1

<table>
<thead>
<tr>
<th>r1</th>
<th>p1</th>
</tr>
</thead>
<tbody>
<tr>
<td>r2</td>
<td>p2</td>
</tr>
<tr>
<td>r3</td>
<td>p3</td>
</tr>
<tr>
<td>r4</td>
<td>p4</td>
</tr>
<tr>
<td>r5</td>
<td>p5</td>
</tr>
</tbody>
</table>

Map table

<table>
<thead>
<tr>
<th>p7</th>
</tr>
</thead>
<tbody>
<tr>
<td>p8</td>
</tr>
<tr>
<td>p9</td>
</tr>
</tbody>
</table>

Free-list
Renaming example

xor r1 ^ r2 → r3
add r3 + r4 → r4
sub r5 - r2 → r3
addi r3 + 1 → r1

xor p1 ^ p2 → p6
add p6 + p4

xor r1 ^ r2 → r3
diri r3 + r4 → r4
sub r5 - r2 → r3
diri r3 + 1 → r1

xor p1 ^ p2 → p6
diri p6 + p4

Map table

r1 p1
r2 p2
r3 p6
r4 p4
r5 p5

Free-list

p7 p8 p10

xor r1 ^ r2 → r3
diri r3 + r4 → r4
sub r5 - r2 → r3
diri r3 + 1 → r1

xor p1 ^ p2 → p6
diri p6 + p4

Map table

r1 p1
r2 p2
r3 p6
r4 p4
r5 p5

Free-list

p7 p8 p10

xor r1 ^ r2 → r3
diri r3 + r4 → r4
sub r5 - r2 → r3
diri r3 + 1 → r1

xor p1 ^ p2 → p6
diri p6 + p4

Map table

r1 p1
r2 p2
r3 p6
r4 p4
r5 p5

Free-list

p7 p8 p10

xor r1 ^ r2 → r3
diri r3 + r4 → r4
sub r5 - r2 → r3
diri r3 + 1 → r1

xor p1 ^ p2 → p6
diri p6 + p4

Map table

r1 p1
r2 p2
r3 p6
r4 p4
r5 p5

Free-list

p7 p8 p10
Renaming example

xor r1 ^ r2 → r3
add r3 + r4 → r4
sub r5 - r2 → r3
addi r3 + 1 → r1

xor p1 ^ p2 → p6
add p6 + p4 → p7
sub p5 - p2 → p8

->

Map table

Free-list

xor r1 ^ r2 → r3
add r3 + r4 → r4
sub r5 - r2 → r3
addi r3 + 1 → r1

xor p1 ^ p2 → p6
add p6 + p4 → p7
sub p5 - p2 → p8

->

Map table

Free-list

- 41 -

Renaming example

xor r1 ^ r2 → r3
add r3 + r4 → r4
sub r5 - r2 → r3
addi r3 + 1 → r1

xor p1 ^ p2 → p6
add p6 + p4 → p7
sub p5 - p2 → p8

->

Map table

Free-list

xor r1 ^ r2 → r3
add r3 + r4 → r4
sub r5 - r2 → r3
addi r3 + 1 → r1

xor p1 ^ p2 → p6
add p6 + p4 → p7
sub p5 - p2 → p8

->

Map table

Free-list

- 43 -
Renaming example

\[\text{xor } r1 \wedge r2 \rightarrow r3\]
\[\text{add } r3 + r4 \rightarrow r4\]
\[\text{sub } r5 - r2 \rightarrow r3\]
\[\text{add } r3 + 1 \rightarrow r1\]

\[\text{xor } p1 \wedge p2 \rightarrow p6\]
\[\text{add } p6 + p4 \rightarrow p7\]
\[\text{sub } p5 - p2 \rightarrow p8\]
\[\text{add } p8 + 1 \rightarrow p9\]

<table>
<thead>
<tr>
<th>r1</th>
<th>p9</th>
</tr>
</thead>
<tbody>
<tr>
<td>r2</td>
<td>p2</td>
</tr>
<tr>
<td>r3</td>
<td>p8</td>
</tr>
<tr>
<td>r4</td>
<td>p7</td>
</tr>
<tr>
<td>r5</td>
<td>p5</td>
</tr>
</tbody>
</table>

Map table

Free-list

Out-of-order Pipeline

Dynamic Scheduling Mechanisms

Dispatch

- Put renamed instructions into out-of-order structures
- Re-order buffer (ROB)
  - Holds instructions until commit
- Issue Queue
  - Central piece of scheduling logic
  - Holds un-executed instructions
  - Tracks ready inputs
    - Physical register names + ready bit
    - “AND” the bits to tell if ready

\[
\begin{array}{cccccccc}
\text{Insn} & \text{Inp1} & R & \text{Inp2} & R & \text{Dst} & \text{Bday} \\
\hline
\end{array}
\]

Ready?
Dispatch Steps

- Allocate Issue Queue (IQ) slot
  - Full? Stall
- Read ready bits of inputs
  - 1-bit per physical reg
- Clear ready bit of output in table
  - Instruction has not produced value yet
- Write data into Issue Queue (IQ) slot

Dispatch Example

```
xor p1 ^ p2 ➜ p6
add p6 + p4 ➜ p7
sub p5 - p2 ➜ p8
add p8 + 1 ➜ p9
```

Issue Queue

```
<table>
<thead>
<tr>
<th>Insn</th>
<th>Inp1</th>
<th>R</th>
<th>Inp2</th>
<th>R</th>
<th>Dst</th>
<th>Bday</th>
</tr>
</thead>
<tbody>
<tr>
<td>xor</td>
<td>p1</td>
<td>y</td>
<td>p2</td>
<td>y</td>
<td>p6</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Ready bits

```
p1 y
p2 y
p3 y
p4 y
p5 y
p6 y
p7 y
p8 y
p9 y
```
Dispatch Example

Insn  Inp1  R  Inp2  R  Dst  Bday
xor   p1   y  p2   y   p6   0
add   p6   n  p4   y   p7   1
sub   p5   y  p2   y   p8   2

Out-of-order pipeline

• Execution (out-of-order) stages
• Select ready instructions
• Send for execution
• Wakeup dependents

Dynamic Scheduling/Issue Algorithm

• Data structures:
  • Ready table[phys_reg] ⇒ yes/no  (part of issue queue)

• Algorithm at “schedule” stage (prior to read registers):
  foreach instruction:
  
  if table[insn.phys_input1] == ready &&
    table[insn.phys_input2] == ready then
    insn is “ready”
  select the oldest “ready” instruction
  table[insn.phys_output] = ready
Issue = Select + Wakeup

- Select oldest of "ready" instructions
  - "xor" is the oldest ready instruction below
  - "xor" and "sub" are the two oldest ready instructions below
- Note: may have resource constraints: i.e. load/store/floating point

<table>
<thead>
<tr>
<th>Insn</th>
<th>Inp1</th>
<th>R</th>
<th>Inp2</th>
<th>R</th>
<th>Dst</th>
<th>Bday</th>
</tr>
</thead>
<tbody>
<tr>
<td>xor</td>
<td>p1</td>
<td>y</td>
<td>p2</td>
<td>y</td>
<td>p6</td>
<td>0</td>
</tr>
<tr>
<td>add</td>
<td>p6</td>
<td>n</td>
<td>p4</td>
<td>y</td>
<td>p7</td>
<td>1</td>
</tr>
<tr>
<td>sub</td>
<td>p5</td>
<td>y</td>
<td>p2</td>
<td>y</td>
<td>p8</td>
<td>2</td>
</tr>
<tr>
<td>addi</td>
<td>p8</td>
<td>n</td>
<td>---</td>
<td>y</td>
<td>p9</td>
<td>3</td>
</tr>
</tbody>
</table>

- Ready!

- Wakeup dependent instructions
  - Search for destination (Dst) in inputs & set "ready" bit
    - Implemented with a special memory array circuit called a Content Addressable Memory (CAM)
  - Also update ready-bit table for future instructions

<table>
<thead>
<tr>
<th>Insn</th>
<th>Inp1</th>
<th>R</th>
<th>Inp2</th>
<th>R</th>
<th>Dst</th>
<th>Bday</th>
</tr>
</thead>
<tbody>
<tr>
<td>xor</td>
<td>p1</td>
<td>y</td>
<td>p2</td>
<td>y</td>
<td>p6</td>
<td>0</td>
</tr>
<tr>
<td>add</td>
<td>p6</td>
<td>y</td>
<td>p4</td>
<td>y</td>
<td>p7</td>
<td>1</td>
</tr>
<tr>
<td>sub</td>
<td>p5</td>
<td>y</td>
<td>p2</td>
<td>y</td>
<td>p8</td>
<td>2</td>
</tr>
<tr>
<td>addi</td>
<td>p8</td>
<td>y</td>
<td>---</td>
<td>y</td>
<td>p9</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Ready bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1 y</td>
</tr>
<tr>
<td>p2 y</td>
</tr>
<tr>
<td>p3 y</td>
</tr>
<tr>
<td>p4 y</td>
</tr>
<tr>
<td>p5 y</td>
</tr>
<tr>
<td>p6 y</td>
</tr>
<tr>
<td>p7 n</td>
</tr>
<tr>
<td>p8 y</td>
</tr>
<tr>
<td>p9 n</td>
</tr>
</tbody>
</table>

- For multi-cycle operations (loads, floating point)
  - Wakeup deferred a few cycles
  - Include checks to avoid structural hazards

Issue

- Select/Wakeup one cycle
- Dependent instructions execute on back-to-back cycles
- Next cycle: add/addi are ready:

<table>
<thead>
<tr>
<th>Insn</th>
<th>Inp1</th>
<th>R</th>
<th>Inp2</th>
<th>R</th>
<th>Dst</th>
<th>Bday</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>p6</td>
<td>y</td>
<td>p4</td>
<td>y</td>
<td>p7</td>
<td>1</td>
</tr>
<tr>
<td>addi</td>
<td>p8</td>
<td>y</td>
<td>---</td>
<td>y</td>
<td>p9</td>
<td>3</td>
</tr>
</tbody>
</table>

- Issued instructions are removed from issue queue
  - Free up space for subsequent instructions

OOO execution (2-wide)
When Does Register Read Occur?

- Current approach: after select, right before execute
  - Not during in-order part of pipeline, in out-of-order part
  - Read physical register (renamed)
  - Or get value via bypassing (based on physical register name)
- This is Pentium 4, MIPS R10k, Alpha 21264, IBM Power4, Intel's "Sandy Bridge" (2011)
- Physical register file may be large
  - Multi-cycle read
- Older approach:
  - Read as part of "issue" stage, keep values in Issue Queue
    - At commit, write them back to "architectural register file"
  - Pentium Pro, Core 2, Core i7
  - Simpler, but may be less energy efficient (more data movement)

Re-order Buffer (ROB)

- ROB entry holds all info for recovery/commit
  - All instructions & in order
  - Architectural register names, physical register names, insn type
  - Not removed until very last thing ("commit")

- Operation
  - Dispatch: insert at tail (if full, stall)
  - Commit: remove from head (if not yet done, stall)

- Purpose: tracking for in-order commit
  - Maintain appearance of in-order execution
  - Done to support:
    - Misprediction recovery
    - Freeing of physical registers

Renaming Revisited
Renaming revisited

- Track (or “log”) the “overwritten register” in ROB
- Free this register at commit
- Also used to restore the map table on “recovery”
  - Branch mis-prediction recovery

Register Renaming Algorithm (Full)

- Two key data structures:
  - maptable[architectural_reg] → physical_reg
- Free list: allocate (new) & free registers (implemented as a queue)
- Algorithm: at “decode” stage for each instruction:
  insn.phys_input1 = maptable[insn.arch_input1]
  insn.phys_input2 = maptable[insn.arch_input2]
  insn.old_phys_output = maptable[insn.arch_output]
  new_reg = new_phys_reg()
  maptable[insn.arch_output] = new_reg
  insn.phys_output = new_reg
- At “commit”
  - Once all older instructions have committed, free register
    free_phys_reg(insn. old_phys_output)

Recovery

- Completely remove wrong path instructions
  - Flush from IQ
  - Remove from ROB
  - Restore map table to before misprediction
  - Free destination registers
- How to restore map table?
  - Option #1: log-based reverse renaming to recover each instruction
    - Tracks the old mapping to allow it to be reversed
    - Done sequentially for each instruction (slow)
    - See next slides
  - Option #2: checkpoint-based recovery
    - Checkpoint state of maptable and free list each cycle
    - Faster recovery, but requires more state
  - Option #3: hybrid (checkpoint for branches, unwind for others)

Renaming example

xor r1 ^ r2 → r3
add r3 + r4 → r4
sub r5 - r2 → r3
addi r3 + 1 → r1

| r1 | p1 |  
|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|
| r2 | p2 |  
| r3 | p3 |  
| r4 | p4 |  
| r5 | p5 |  

Map table

| p6 | p7 | p8 | p9 | p10 |
|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|

Free-list
Renaming example

xor r1 ^ r2 → r3
add r3 + r4 → r4
sub r5 - r2 → r3
addi r3 + 1 → r1

xor p1 ^ p2 →
add p6 + p4 →
sub p7 - p2 → p6
addi p8 + 1 → p3

[ p3 ]
[ p4 ]

r1 p1
r2 p2
r3 p3
r4 p4
r5 p5
p6
p7
p8
p9
p10
Map table Free-list

Renaming example

xor r1 ^ r2 → r3
add r3 + r4 → r4
sub r5 - r2 → r3
addi r3 + 1 → r1

xor p1 ^ p2 →
add p6 + p4 →
sub p7 - p2 → p6
addi p8 + 1 → p3

[ p3 ]
[ p4 ]

r1 p1
r2 p2
r3 p3
r4 p4
r5 p5
p6
p7
p8
p9
p10
Map table Free-list
Renaming example

\[
xor \ r1 \ ^ \ r2 \rightarrow \ r3 \\
add \ r3 \ + \ r4 \rightarrow \ r4 \\
sub \ r5 - r2 \rightarrow \ r3 \\
addi \ r3 \ + \ 1 \rightarrow \ r1
\]

\[
xor \ p1 \ ^ \ p2 \rightarrow \ p6 \\
add \ p6 \ + \ p4 \rightarrow \ p7 \\
sub \ p5 - p2 \rightarrow \ p8 \\
addi \ p8 + 1 \rightarrow \ p1
\]

- 77 -

CS 740 F'14

Renaming example

\[
xor \ r1 \ ^ \ r2 \rightarrow \ r3 \\
add \ r3 \ + \ r4 \rightarrow \ r4 \\
sub \ r5 - r2 \rightarrow \ r3 \\
addi \ r3 \ + \ 1 \rightarrow \ r1
\]

\[
xor \ p1 \ ^ \ p2 \rightarrow \ p6 \\
add \ p6 \ + \ p4 \rightarrow \ p7 \\
sub \ p5 - p2 \rightarrow \ p8 \\
addi \ p8 + 1 \rightarrow \ p1
\]

- 78 -

CS 740 F'14

Renaming example

\[
xor \ r1 \ ^ \ r2 \rightarrow \ r3 \\
add \ r3 \ + \ r4 \rightarrow \ r4 \\
sub \ r5 - r2 \rightarrow \ r3 \\
addi \ r3 \ + \ 1 \rightarrow \ r1
\]

\[
xor \ p1 \ ^ \ p2 \rightarrow \ p6 \\
add \ p6 \ + \ p4 \rightarrow \ p7 \\
sub \ p5 - p2 \rightarrow \ p8 \\
addi \ p8 + 1 \rightarrow \ p1
\]

- 79 -

CS 740 F'14

Renaming example

\[
xor \ r1 \ ^ \ r2 \rightarrow \ r3 \\
add \ r3 \ + \ r4 \rightarrow \ r4 \\
sub \ r5 - r2 \rightarrow \ r3 \\
addi \ r3 \ + \ 1 \rightarrow \ r1
\]

\[
xor \ p1 \ ^ \ p2 \rightarrow \ p6 \\
add \ p6 \ + \ p4 \rightarrow \ p7 \\
sub \ p5 - p2 \rightarrow \ p8 \\
addi \ p8 + 1 \rightarrow \ p1
\]

- 80 -

CS 740 F'14
Recovery Example

Now, let's use this info. to recover from a branch misprediction.

```
bnz r1, loop
xor r1 ^ r2 -> r3
add r3 + r4 -> r4
sub r5 - r2 -> r3
addi r3 + 1 -> r1

bnz p1, loop
xor p1 ^ p2 -> p6
add p6 + p4 -> p7
sub p5 - p2 -> p8
addi p8 + 1 -> p9
```

Map table

```
r1  p9
r2  p2
r3  p8
r4  p7
r5  p5
```

Free-list

```
[  ]
[ p3 ]
[ p4 ]
[ p6 ]
[ p1 ]
```

Recovery Example

```
bnz r1, loop
xor r1 ^ r2 -> r3
add r3 + r4 -> r4
sub r5 - r2 -> r3
addi r3 + 1 -> r1

bnz p1, loop
xor p1 ^ p2 -> p6
add p6 + p4 -> p7
sub p5 - p2 -> p8
addi p8 + 1 -> p9
```

Map table

```
r1  p1
r2  p2
r3  p8
r4  p7
r5  p5
```

Free-list

```
[  ]
[ p3 ]
[ p4 ]
[ p6 ]
[ p1 ]
```

Recovery Example

```
bnz r1, loop
xor r1 ^ r2 -> r3
add r3 + r4 -> r4
sub r5 - r2 -> r3
addi r3 + 1 -> r1

bnz p1, loop
xor p1 ^ p2 -> p6
add p6 + p4 -> p7
sub p5 - p2 -> p8
addi p8 + 1 -> p9
```

Map table

```
r1  p1
r2  p2
r3  p6
r4  p7
r5  p5
```

Free-list

```
[  ]
[ p3 ]
[ p4 ]
[ p6 ]
[ p1 ]
```

Recovery Example

```
bnz r1, loop
xor r1 ^ r2 -> r3
add r3 + r4 -> r4
sub r5 - r2 -> r3
addi r3 + 1 -> r1

bnz p1, loop
xor p1 ^ p2 -> p6
add p6 + p4 -> p7
sub p5 - p2 -> p8
addi p8 + 1 -> p9
```

Map table

```
r1  p1
r2  p2
r3  p8
r4  p4
r5  p5
```

Free-list

```
[  ]
[ p3 ]
[ p4 ]
[ p6 ]
[ p1 ]
```

Recovery Example

```
bnz r1, loop
xor r1 ^ r2 -> r3
add r3 + r4 -> r4
sub r5 - r2 -> r3
addi r3 + 1 -> r1

bnz p1, loop
xor p1 ^ p2 -> p6
add p6 + p4 -> p7
sub p5 - p2 -> p8
addi p8 + 1 -> p9
```

Map table

```
r1  p1
r2  p2
r3  p8
r4  p4
r5  p5
```

Free-list

```
[  ]
[ p3 ]
[ p4 ]
[ p6 ]
[ p1 ]
```

Recovery Example

```
bnz r1, loop
xor r1 ^ r2 -> r3
add r3 + r4 -> r4
sub r5 - r2 -> r3
addi r3 + 1 -> r1

bnz p1, loop
xor p1 ^ p2 -> p6
add p6 + p4 -> p7
sub p5 - p2 -> p8
addi p8 + 1 -> p9
```

Map table

```
r1  p1
r2  p2
r3  p8
r4  p4
r5  p5
```

Free-list

```
[  ]
[ p3 ]
[ p4 ]
[ p6 ]
[ p1 ]
```
Recovery Example

```plaintext
bnz r1, loop
xor r1 ^ r2 ➔ r3
```

```plaintext
bnz p1, loop
xor p1 ^ p2 ➔ p6
```

```
[     ]
```

```
[ p3 ]
```

```
```

```plaintext
bnz r1, loop
xor r1 ^ r2 ➔ r3
```

```plaintext
bnz p1, loop
xor p1 ^ p2 ➔ p6
```

```
[     ]
```

```
[ p3 ]
```

```
```

Commit

- P3 was r3 before xor
- P6 is r3 after xor
- Anything older than xor should read p3
- Anything younger than xor should read p6 (until another insn writes r3)
- At commit of xor, no older instructions exist

Freeing over-written register

- P3 was r3 before xor
- P6 is r3 after xor
- Anything older than xor should read p3
- Anything younger than xor should read p6 (until another insn writes r3)
- At commit of xor, no older instructions exist
Commit Example

xor r1 ^ r2 ➜ r3
add r3 + r4 ➜ r4
sub r5 - r2 ➜ r3
addi r3 + 1 ➜ r1

xor p1 ^ p2 ➜ p6
add p6 + p4 ➜ p7
sub p5 - p2 ➜ p8
addi p8 + 1 ➜ p9

r1  p9
r2  p2
r3  p8
r4  p7
r5  p5

Map table

p10

Free-list
Commit Example

addi r3 + 1 → r1
addi p8 + 1 → p9

Map table

<table>
<thead>
<tr>
<th>r1</th>
<th>p9</th>
</tr>
</thead>
<tbody>
<tr>
<td>r2</td>
<td>p2</td>
</tr>
<tr>
<td>r3</td>
<td>p8</td>
</tr>
<tr>
<td>r4</td>
<td>p7</td>
</tr>
<tr>
<td>r5</td>
<td>p5</td>
</tr>
</tbody>
</table>

Free-list

<table>
<thead>
<tr>
<th>p10</th>
</tr>
</thead>
<tbody>
<tr>
<td>p3</td>
</tr>
<tr>
<td>p4</td>
</tr>
<tr>
<td>p6</td>
</tr>
</tbody>
</table>

Out-of-order Pipeline

- Fetch
- Decode
- Rename
- Dispatch
- Issue
- Reg-read
- Execute
- Writeback
- Commit

Have unique register names
Now put into out-of-order execution structures

Out-of-Order: Benefits & Challenges
Dynamic Scheduling Operation (Recap)

- Dynamic scheduling
  - Totally in the hardware (not visible to software)
  - Also called "out-of-order execution" (OoO)
- Fetch many instructions into instruction window
  - Use branch prediction to speculate past (multiple) branches
  - Flush pipeline on branch misprediction
- Rename registers to avoid false dependencies
- Execute instructions as soon as possible
  - Register dependencies are known
  - Handling memory dependencies is harder
- "Commit" instructions in order
  - Anything strange happens before commit, just flush the pipeline
- How much out-of-order? Core i7 "Sandy Bridge":
  - 168-entry reorder buffer, 160 integer registers, 54-entry scheduler

i486 Pipeline

- Fetch
  - Load 16-bytes of instruction into prefetch buffer
- Decode1
  - Determine instruction length, instruction type
- Decode2
  - Compute memory address
  - Generate immediate operands
- Execute
  - Register Read
  - ALU operation
  - Memory read/write
- Write-Back
  - Update register file

Pipeline Stage Details

- Fetch
  - Moves 16 bytes of instruction stream into code queue
  - Not required every time
    - About 5 instructions fetched at once
    - Only useful if don't branch
  - Avoids need for separate instruction cache
- D1
  - Determine total instruction length
    - Signals code queue aligner where next instruction begins
  - May require two cycles
    - When multiple operands must be decoded
    - About 6% of "typical" DOS program
- D2
  - Extract memory displacements and immediate operands
  - Compute memory addresses
    - Add base register, and possibly scaled index register
  - May require two cycles
    - If index register involved, or both address & immediate operand
    - Approx. 5% of executed instructions
- EX
  - Read register operands
  - Compute ALU function
  - Read or write memory (data cache)
- WB
  - Update register result

Stage Details (Cont.)
Data Hazards

- Data Hazards
  - Generated Used Handling
  - ALU ALU EX-EX Forwarding
  - Load ALU EX-EX Forwarding
  - ALU Store EX-EX Forwarding
  - ALU Eff. Address (Stall) + EX-ID2 Forwarding

Control Hazards

- Jump Instruction Processing
  - Continue pipeline assuming branch not taken
  - Resolve branch condition in EX stage
  - Also speculatively fetch at target during EX stage

Control Hazards (Cont.)

Branch Not Taken
- Allow pipeline to continue.
- Total of 1 cycle for instruction

Branch taken
- Flush instructions in pipe
- Begin ID1 at target.
- Total of 3 cycles for instruction

Comparison to 386

- Cycles Per Instruction
  - Instruction Type
  - 386 Cycles
  - 486 Cycles
  - Load 4 1
  - Store 2 1
  - ALU 2 1
  - Jump taken 9 3
  - Jump not taken 3 1
  - Call 9 3

- Reasons for Improvement
  - On chip cache
    - Faster loads & stores
  - More pipelining
Pentium Block Diagram

Pentium Pipeline

Pentium Pro (P6)

- History
  - Announced in Feb. ’95
  - Delivering in high end machines now

- Features
  - Dynamically translates instructions to more regular format
    - Very wide RISC instructions
  - Executes operations in parallel
    - Up to 5 at once
  - Very deep pipeline
    - 12-18 cycle latency

Superscalar Execution

- Can Execute Instructions I1 & I2 in Parallel if:
  - Both are "simple" instructions
    - Don’t require microcode sequencing
    - Some operations require U-pipe resources
    - 90% of SpecInt instructions
  - I1 is not a jump
  - Destination of I1 not source of I2
    - But can handle I1 setting CC and I2 being cond. jump
  - Destination of I1 not destination of I2
  - If Conditions Don’t Hold
    - Issue I1 to U Pipe
    - I2 issued on next cycle
      - Possibly paired with following instruction
Branch Prediction

- **Branch Target Buffer**
  - Stores information about previously executed branches
    - Indexed by instruction address
    - Specifies branch destination + whether or not taken
  - 256 entries

- **Branch Processing**
  - Look for instruction in BTB
  - If found, start fetching at destination
  - Branch condition resolved early in WB
    - If prediction correct, no branch penalty
    - If prediction incorrect, lose ~3 cycles
      - Which corresponds to > 3 instructions
  - Update BTB

Core i7 Pipeline: Big Picture

Core i7 Pipeline: Front End
OoO Execution is all around us

• Qualcomm Krait processor (in phones)
  • based on ARM Cortex A15 processor
  • out-of-order 1.5GHz dual-core
  • 3-wide fetch/decode
  • 4-wide issue
  • 11-stage integer pipeline
  • 28nm process technology

Out of Order: Benefits

• Allows speculative re-ordering
  • Loads / stores
  • Branch prediction to look past branches
• Done by hardware
  • Compiler may want different schedule for different hw configs
  • Hardware has only its own configuration to deal with
• Schedule can change due to cache misses
• Memory-level parallelism
  • Executes "around" cache misses to find independent instructions
  • Finds and initiates independent misses, reducing memory latency
  • Especially good at hiding L2 hits (~12 cycles in Core i7)
**Challenges for Out-of-Order Cores**

- Design complexity
  - More complicated than in-order? Certainly!
  - But, we have managed to overcome the design complexity
- Clock frequency
  - Can we build a "high ILP" machine at high clock frequency?
  - Yep, with some additional pipe stages, clever design
- Limits to (efficiently) scaling the window and ILP
  - Large physical register file
  - Fast register renaming/wakeup/select/load queue/store queue
  - Branch & memory depend. prediction (limits effective window size)
    - 95% branch mis-prediction: 1 in 20 branches, or 1 in 100 insn.
  - Plus all the issues of building "wide" in-order superscalar
- Power efficiency
  - Today, even mobile phone chips are out-of-order cores

**Architectural Performance**

- Metric
  - SpecX92/Mhz: Normalizes with respect to clock speed
- Sampling
  
<table>
<thead>
<tr>
<th>Processor</th>
<th>MHz</th>
<th>SpecInt92</th>
<th>IntAP</th>
<th>SpecFP92</th>
<th>FltAP</th>
</tr>
</thead>
<tbody>
<tr>
<td>i386/387</td>
<td>33</td>
<td>6</td>
<td>0.2</td>
<td>3</td>
<td>0.1</td>
</tr>
<tr>
<td>i486DX</td>
<td>50</td>
<td>28</td>
<td>0.6</td>
<td>13</td>
<td>0.3</td>
</tr>
<tr>
<td>Pentium</td>
<td>150</td>
<td>181</td>
<td>1.2</td>
<td>125</td>
<td>0.8</td>
</tr>
<tr>
<td>PentiumPro</td>
<td>200</td>
<td>320</td>
<td>1.6</td>
<td>283</td>
<td>1.4</td>
</tr>
<tr>
<td>MIPS R3000A25</td>
<td>16.1</td>
<td>0.6</td>
<td>21.7</td>
<td>0.9</td>
<td></td>
</tr>
<tr>
<td>MIPS R10000200</td>
<td>300</td>
<td>1.5</td>
<td>600</td>
<td>3.0</td>
<td></td>
</tr>
<tr>
<td>Alpha 21164a417</td>
<td>500</td>
<td>1.2</td>
<td>750</td>
<td>1.8</td>
<td></td>
</tr>
</tbody>
</table>

**i486 Pipeline**

- Fetch
  - Load 16-bytes of instruction into prefetch buffer
- Decode1
  - Determine instruction length, instruction type
- Decode2
  - Compute memory address
  - Generate immediate operands
- Execute
  - Register Read
  - ALU operation
  - Memory read/write
- Write-Back

**Pipeline Stage Details**

- Fetch
  - Moves 16 bytes of instruction stream into code queue
  - Not required every time
    - About 5 instructions fetched at once
    - Only useful if don’t branch
  - Avoids need for separate instruction cache
- D1
  - Determine total instruction length
    - Signals code queue aligner where next instruction begins
  - May require two cycles
    - When multiple operands must be decoded
    - About 6% of "typical" DOS program
Stage Details (Cont.)

- D2
  - Extract memory displacements and immediate operands
  - Compute memory addresses
    - Add base register, and possibly scaled index register
  - May require two cycles
    - If index register involved, or both address & immediate operand
    - Approx. 5% of executed instructions

- EX
  - Read register operands
  - Compute ALU function
  - Read or write memory (data cache)

Data Hazards

- Data Hazards
  - Generated Used Handling
  - ALU ALU EX-EX Forwarding
  - Load ALU EX-EX Forwarding
  - ALU Store EX-EX Forwarding
  - ALU Eff. Address (Stall) + EX-ID2 Forwarding

Control Hazards

- Jump Instruction Processing
  - Continue pipeline assuming branch not taken
  - Resolve branch condition in EX stage
  - Also speculatively fetch at target during EX stage

Control Hazards (Cont.)

- Branch Not Taken
  - Allow pipeline to continue.
  - Total of 1 cycle for instruction

- Branch taken
  - Flush instructions in pipe
  - Begin ID1 at target.
  - Total of 3 cycles for instruction
Comparison with Our pAlpha Pipeline

- Two Decoding Stages
  - Harder to decode CISC instructions
  - Effective address calculation in D2
- Multicycle Decoding Stages
  - For more difficult decodings
  - Stalls incoming instructions
- Combined Mem/EX Stage
  - Avoids load stall without load delay slot
    - But introduces stall for address computation

Comparison to 386

- Cycles Per Instruction
  
<table>
<thead>
<tr>
<th>Instruction Type</th>
<th>386 Cycles</th>
<th>486 Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>Store</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>ALU</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Jump taken</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>Jump not taken</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Call</td>
<td>9</td>
<td>3</td>
</tr>
</tbody>
</table>

- Reasons for Improvement
  - On chip cache
    - Faster loads & stores
  - More pipelining

Pentium Block Diagram

Pentium Pipeline
**Superscalar Execution**

- Can Execute Instructions I1 & I2 in Parallel if:
  - Both are "simple" instructions
    - Don't require microcode sequencing
    - Some operations require U-pipe resources
    - 90% of SpecInt instructions
  - I1 is not a jump
  - Destination of I1 not source of I2
    - But can handle I1 setting CC and I2 being cond. jump
  - Destination of I1 not destination of I2
- If Conditions Don't Hold
  - Issue I1 to U Pipe
  - I2 issued on next cycle
    - Possibly paired with following instruction

**Branch Prediction**

- Branch Target Buffer
  - Stores information about previously executed branches
    - Indexed by instruction address
    - Specifies branch destination + whether or not taken
  - 256 entries
- Branch Processing
  - Look for instruction in BTB
  - If found, start fetching at destination
  - Branch condition resolved early in WB
    - If prediction correct, no branch penalty
    - If prediction incorrect, lose ~3 cycles
      » Which corresponds to > 3 instructions

**Superscalar Terminology**

- Basic
  - *Superscalar* Able to issue > 1 instruction / cycle
  - *Superpipelined* Deep, but not superscalar pipeline.
    - E.g., MIPS R5000 has 8 stages
  - *Branch prediction* Logic to guess whether or not branch will be taken, and possibly branch target
- Advanced
  - *Out-of-order* Able to issue instructions out of program order
  - *Speculation* Execute instructions beyond branch points, possibly nullifying later
  - *Register renaming* Able to dynamically assign physical registers to instructions
  - *Retire unit* Logic to keep track of instructions as

**Superscalar Execution Example**

- Assumptions
  - Single FP adder takes 2 cycles
  - Single FP multiplier takes 5 cycles
  - Can issue add & multiply together (in order)
  - Must issue in-order
  - Critical Path = 9 cycles

**Data Flow**

- v: `addt $f2, $f4, $f10`
- w: `mult $f10, $f6, $f10`
- x: `addt $f10, $f8, $f12`
- y: `addt $f4, $f6, $f12`
- z: `addt $f4, $f8, $f10`

- Critical Path = 9 cycles
Adding Advanced Features

- Out Of Order Issue
  - Can start \( y \) as soon as adder available
  - Must hold back \( z \) until \( \$f10 \) not busy & adder available

\[
\begin{align*}
v &: \text{addt} \ $f2, \ $f4, \ $f10 \\
vw &: \text{mult} \ $f10, \ $f6, \ $f10 \\
vwx &: \text{addt} \ $f10, \ $f8, \ $f12 \\
vwx y &: \text{addt} \ $f4, \ $f6, \ $f4 \\
vwx yz &: \text{addt} \ $f4, \ $f8, \ $f10
\end{align*}
\]

- With Register Renaming

\[
\begin{align*}
v &: \text{addt} \ $f2, \ $f4, \ $f10a \\
vw &: \text{mult} \ $f10a, \ $f6, \ $f10a \\
vwx &: \text{addt} \ $f10a, \ $f8, \ $f12 \\
vwx y &: \text{addt} \ $f4, \ $f6, \ $f4 \\
vwx yz &: \text{addt} \ $f4, \ $f8, \ $f10
\end{align*}
\]

Pentium Pro (P6)

- History
  - Announced in Feb. ’95
  - Delivering in high end machines now

- Features
  - Dynamically translates instructions to more regular format
    - Very wide RISC instructions
  - Executes operations in parallel
    - Up to 5 at once
  - Very deep pipeline
    - 12-18 cycle latency

PentiumPro Block Diagram

Dynamically Scheduled Processors

- Fetch & graduate in-order, issue out-of-order

\[
\begin{align*}
0x1c &: b = c / 3; \\
0x18 &: z = a + 2; \\
0x14 &: y = x + 1; \\
0x10 &: x = *p;
\end{align*}
\]
**PentiumPro Operation**
- Translates instructions dynamically into “Uops”
  - 118 bits wide
  - Holds operation, two sources, and destination
- Executes Uops with “Out of Order” engine
  - Uop executed when
    - Operands available
    - Functional unit available
- Execution controlled by “Reservation Stations”
  - Keeps track of data dependencies between uops
  - Allocates resources

**Read-after-Write (RAW) Dependences**
- Also known as a “true” dependence
- Example:
  
  S1: addq r1, r2, r3  
  S2: addq r3, r4, r4

- How to optimize?
  - cannot be optimized away

**Write-after-Read (WAR) Dependences**
- Also known as an “anti” dependence
- Example:
  
  S1: addq r1, r2, r3  
  S2: addq r4, r5, r3  
  ...  
  addq r1, r6, r7

- How to optimize?
  - rename dependent register (e.g., r1 in S2 -> r8)
    
    S1: addq r1, r2, r3  
    S2: addq r4, r5, r8  
    ...  
    addq r8, r6, r7

**Write-after-Write (WAW) Dependences**
- Also known as an “output” dependence
- Example:
  
  S1: addq r1, r2, r3  
  S2: addq r4, r5, r3\(\uparrow\)  
  ...  
  addq r3, r6, r7

- How to optimize?
  - rename dependent register (e.g., r3 in S2 \(\rightarrow\) r8)
    
    S1: addq r1, r2, r3  
    S2: addq r4, r5, r8  
    ...  
    addq r8, r6, r7
Living with Expensive Branches

- Mispredicted Branch Carries a High Cost
  - Must flush many in-flight instructions
  - Start fetching at correct target
  - Will get worse with deeper and wider pipelines
- Impact on Programmer / Compiler
  - Avoid conditionals when possible
    - Bit manipulation tricks
  - Use special conditional-move instructions
    - Recent additions to many instruction sets
  - Make branches predictable
    - Very low overhead when predicted correctly

Branch Prediction Example

```c
static void loop1() {
    int i;
data_t abs_sum = (data_t) 0;
data_t prod = (data_t) 1;
for (i = 0; i < CNT; i++) {
data_t x = dat[i];data_t ax;
    ax = ABS(x);
    abs_sum += ax;
    prod *= x;
}answer = abs_sum+prod;
}
```

MIPS Code

```assembly
# define ABS(x) x < 0 ? -x : x
```

Branch Prediction Example

- Compute sum of absolute values
- Compute product of original values

Some Interesting Patterns

- PPPPPPPPP
  - Should give perfect prediction
  - Will mispredict 1/2 of the time
- RRRRRRRRR
  - Should alternate between states No! and No?
- N*N[P[PNPN]]
  - Should alternate between states No? and Yes?

Loop Performance (FP)

<table>
<thead>
<tr>
<th>Pattern</th>
<th>R3000</th>
<th>PPC 604</th>
<th>Pentium</th>
</tr>
</thead>
<tbody>
<tr>
<td>PPPPPPPPP</td>
<td>13.6</td>
<td>9.2</td>
<td>21.1</td>
</tr>
<tr>
<td>RRRRRRRRR</td>
<td>13.6</td>
<td>12.6</td>
<td>3.4</td>
</tr>
<tr>
<td>N*N[P[PNPN]]</td>
<td>13.6</td>
<td>15.8</td>
<td>6.6</td>
</tr>
<tr>
<td>N*P[PPPN]</td>
<td>13.6</td>
<td>15.9</td>
<td>6.7</td>
</tr>
<tr>
<td>N*P[PPPN]</td>
<td>13.6</td>
<td>12.5</td>
<td>3.3</td>
</tr>
<tr>
<td>N*N[P[PNPN]]</td>
<td>13.6</td>
<td>12.5</td>
<td>3.3</td>
</tr>
</tbody>
</table>

- Observations
  - 604 has prediction rates 0%, 50%, and 100%
    - Expected 50% from N*N[P[PNPN]]
    - Expected 25% from N*N[PPNN]
    - Loop so tight that speculate through single branch twice?
  - Pentium appears to be more variable, ranging 0 to 100%
**Loop 1 Surprises**

- Pentium II
  - Random shows clear penalty
  - But others do well
    - More clever prediction algorithm
- R10000
  - Has special "conditional move" instructions
  - Compiler translates $a = \text{Cond} ? \text{Texpr : Fexpr}$ into
    
    ```
    a = Fexpr
    temp = Texpr
    ```

---

**P6 Branch Prediction**

- Two-Level Scheme
  - Yeh & Patt, ISCA ’93
  - Keep shift register showing past $k$ outcomes for branch
  - Use to index $2^k$ entry table
  - Each entry provides 2-bit, saturating counter

---

**Branch Prediction Comparisons**

---

**21264 Branch Prediction Logic**

- Purpose: Predict whether or not branch taken
- 35Kb of prediction information
- 2% of total die size
- Claim 0.7--1.0% misprediction