Pipeline Architecture since 1985

Optional lecture
Clicker questions don’t count
Not on Exams or HW
To learn more, see CS433

No handout but...
Today’s lecture

- Last time, we completed the 5-stage pipeline MIPS.
  - Processors like this were first shipped around 1985
  - Still a fundamentally solid design

- Nevertheless, there have been advances in the past 30 years.
  - Deeper Pipelines
  - Dynamic Branch Prediction
  - Branch Target Buffers (removing the taken branch penalty)
  - Multiple Issue / Superscalar
  - Out-of-order Scheduling
Takeaway points

- The 5-stage pipeline is not a bad mental model for SW developers:
  - Integer arithmetic is cheap
  - Loads can be relatively expensive
    - Especially if there is not other work to be done (e.g., linked list traversals)
  - Branches can be relatively expensive
    - But, primarily if they are not predictable

- In addition, try to avoid long serial dependences; given double D[10]
  - Is faster than:
    - \((((((D[0] + D[1]) + D[2]) + D[3]) + D[4]) + D[5]) + D[6]) + D[7]\)

- There is phenomenal engineering in modern processors
Pipelining can reduce clock cycle time

- Make things faster by making any component smaller!!

\[
\text{CPU time}_{X,P} = \text{Instructions executed}_p \times \text{CPI}_{X,P} \times \text{Clock cycle time}_{X}
\]

Hardware can affect these
“Superpipeling” (if some pipeline stages are good, more must be better! Right?)

MIPS R4000
More Superpipelining

<table>
<thead>
<tr>
<th>Basic Pentium III Processor Misprediction Pipeline</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Basic Pentium 4 Processor Misprediction Pipeline</th>
</tr>
</thead>
</table>
# Historical data from Intel’s processors

Pipeline depths and frequency at introduction.

<table>
<thead>
<tr>
<th>Microprocessor</th>
<th>Year</th>
<th>Clock Rate</th>
<th>Pipeline Stages</th>
</tr>
</thead>
<tbody>
<tr>
<td>i486</td>
<td>1989</td>
<td>25 MHz</td>
<td>5</td>
</tr>
<tr>
<td>Pentium</td>
<td>1993</td>
<td>66 MHz</td>
<td>5</td>
</tr>
<tr>
<td>Pentium Pro</td>
<td>1997</td>
<td>200 MHz</td>
<td>10</td>
</tr>
<tr>
<td>P4 Willamette</td>
<td>2001</td>
<td>2000 MHz</td>
<td>22</td>
</tr>
<tr>
<td>P4 Prescott</td>
<td>2004</td>
<td>3600 MHz</td>
<td>31</td>
</tr>
<tr>
<td>Core 2 Conroe</td>
<td>2006</td>
<td>2930 MHz</td>
<td>14</td>
</tr>
<tr>
<td>Core 2 Yorkfield</td>
<td>2008</td>
<td>2930 MHz</td>
<td>16</td>
</tr>
<tr>
<td>Core i7 Gulftown</td>
<td>2010</td>
<td>3460 MHz</td>
<td>16</td>
</tr>
</tbody>
</table>

What Happened?
Deeper pipelines consume more power

<table>
<thead>
<tr>
<th>Microprocessor</th>
<th>Year</th>
<th>Clock Rate</th>
<th>Pipeline Stages</th>
<th>Power</th>
</tr>
</thead>
<tbody>
<tr>
<td>i486</td>
<td>1989</td>
<td>25 MHz</td>
<td>5</td>
<td>5W</td>
</tr>
<tr>
<td>Pentium</td>
<td>1993</td>
<td>66 MHz</td>
<td>5</td>
<td>10W</td>
</tr>
<tr>
<td>Pentium Pro</td>
<td>1997</td>
<td>200 MHz</td>
<td>10</td>
<td>29W</td>
</tr>
<tr>
<td>P4 Willamette</td>
<td>2001</td>
<td>2000 MHz</td>
<td>22</td>
<td>75W</td>
</tr>
<tr>
<td>P4 Prescott</td>
<td>2004</td>
<td>3600 MHz</td>
<td>31</td>
<td><strong>103W</strong></td>
</tr>
<tr>
<td>Core 2 Conroe</td>
<td>2006</td>
<td>2930 MHz</td>
<td>14</td>
<td>75W</td>
</tr>
<tr>
<td>Core 2 Yorkfield</td>
<td>2008</td>
<td>2930 MHz</td>
<td>16</td>
<td>95W</td>
</tr>
<tr>
<td>Core i7 Gulftown</td>
<td>2010</td>
<td>3460 MHz</td>
<td>16</td>
<td>130W</td>
</tr>
</tbody>
</table>

- Two additional effects:
  - Diminishing returns: pipeline register latency becomes significant
  - Negatively impacts CPI (longer stalls, more instructions flushed)
Mitigating CPI loss 1: Dynamic Branch Prediction

- “Predict not-taken” is cheap, but
  - Some branches are almost always taken
    - Like loop back edges.

- What fraction of time will the highlighted branch mispredict?

```java
for (int i = 0 ; i < 1000 ; i ++) {
    j = 0;
    do {
        // do something
        j++;
    } while (j < 10)
}
```

a) 0%  b) 10%  c) 20%  d) 90%  e) 100%
We can use past behavior to predict future behavior

- Keep 1 bit per branch that remembers the last outcome
  - 0 -> Not taken
  - 1 -> taken

- What fraction of time will the highlighted branch mispredict?

```
for (int i = 0; i < 1000; i++) {
    j = 0;
    do {
        // do something
        j++
    } while (j < 10)
}
```

a) 0%  b) 10%  c) 20%  d) 90%  e) 100%
Adding longer memory can create more stable predictions

- Use a *saturating* 2-bit counter:
  - Increment when branch taken
  - Decrement when branch not-taken
  - Use **top bit** as prediction

- How often will the branch mispredict?

  \[
  \text{a) } 0\% \quad \text{b) } 10\% \quad \text{c) } 20\% \quad \text{d) } 90\% \quad \text{e) } 100\%
  \]
Branch prediction tables

- Too expensive to keep 2 bits per branch in the program
- Instead keep a fixed sized table in the processor
  - Say 1024 2-bit counters.

- “Hash” the program counter (PC) to construct an index:
  - Index = (PC >> 2) ^ (PC >> 12)

- Multiple branches will map to the same entry (interference)
  - But generally not at the same time
    - Programs tend to have working sets.
When to predict branches?

- Need:
  - PC (to access predictor)
  - To know it is a branch (must have decoded the instruction)
  - The branch target (computed from the instruction bits)

- How many flushes on a not taken prediction?
- How many flushes on a taken prediction?
- Is this the best we can do?
Mitigating CPI loss 1: Branch Target Buffers

- Need:
  - PC: Already have at fetch.
  - To know it is a branch
  - The branch target: Can remember and make available at fetch

- Create a table: **Branch Target Buffer**

<table>
<thead>
<tr>
<th>PC</th>
<th>2-bit counter</th>
<th>target</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0010</td>
<td></td>
<td>not started</td>
</tr>
<tr>
<td>x0020</td>
<td>strongly</td>
<td>end loop</td>
</tr>
</tbody>
</table>

- Allocate an entry whenever a branch is taken (& not already present)
BTB accessed in parallel with reading the instruction

- If matching entry found, and ...
- 2-bit counter predicts taken
  - Redirect fetch to branch target
  - Instead of PC+4

What is the taken branch penalty?
- (i.e., how many flushes on a predicted taken branch?)
  - a) 0
  - b) 1
  - c) 2
Removing stalls & flushes can bring CPI down to 1, but there are ways to bring it lower

\[
\text{CPU time}_{X,P} = \text{Instructions executed}_p \times \text{CPI}_{X,P} \times \text{Clock cycle time}_X
\]
Multiple Issue executes multiple instructions in parallel on the same processor
### Issue width has increased over time

<table>
<thead>
<tr>
<th>Microprocessor</th>
<th>Year</th>
<th>Clock Rate</th>
<th>Pipeline Stages</th>
<th>Issue width</th>
</tr>
</thead>
<tbody>
<tr>
<td>i486</td>
<td>1989</td>
<td>25 MHz</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>Pentium</td>
<td>1993</td>
<td>66 MHz</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>Pentium Pro</td>
<td>1997</td>
<td>200 MHz</td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>P4 Willamette</td>
<td>2001</td>
<td>2000 MHz</td>
<td>22</td>
<td>3</td>
</tr>
<tr>
<td>P4 Prescott</td>
<td>2004</td>
<td>3600 MHz</td>
<td>31</td>
<td>3</td>
</tr>
<tr>
<td>Core 2 Conroe</td>
<td>2006</td>
<td>2930 MHz</td>
<td>14</td>
<td>4</td>
</tr>
<tr>
<td>Core 2 Yorkfield</td>
<td>2008</td>
<td>2930 MHz</td>
<td>16</td>
<td>4</td>
</tr>
<tr>
<td>Core i7 Gulftown</td>
<td>2010</td>
<td>3460 MHz</td>
<td>16</td>
<td>4</td>
</tr>
</tbody>
</table>
Static Multiple Issue

- Compiler groups instructions into *issue packets*
  - Group of instructions that can be issued on a single cycle
  - Determined by pipeline resources required

- Think of an issue packet as a very long instruction
  - Specifies multiple concurrent operations

- Compiler must remove some/all hazards
  - Reorder instructions into issue packets
  - *No* dependencies within a packet
  - Pad with nop if necessary
Example: MIPS with Static Dual Issue

- Dual-issue packets
  - One ALU/branch instruction
  - One load/store instruction
  - 64-bit aligned
    - ALU/branch, then load/store
    - Pad an unused instruction with nop

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction type</th>
<th>Pipeline Stages</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>ALU/branch</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>n + 4</td>
<td>Load/store</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>n + 8</td>
<td>ALU/branch</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>n + 12</td>
<td>Load/store</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>n + 16</td>
<td>ALU/branch</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>n + 20</td>
<td>Load/store</td>
<td>IF ID EX MEM WB</td>
</tr>
</tbody>
</table>
Hazards in the Dual-Issue MIPS

- More instructions executing in parallel
- EX data hazard
  - Forwarding avoided stalls with single-issue
  - Now can’t use ALU result in load/store in same packet
    - \texttt{add \ $t0, \ $s0, \ $s1}
    - \texttt{load \ $s2, \ 0($t0)}
    - Split into two packets, effectively a stall
- Load-use hazard
  - Still one cycle use latency, but now two instructions
- More aggressive scheduling required
Scheduling Example

Loop: lw $t0, 0($s1)      # $t0=array element
      addu $t0, $t0, $s2    # add scalar in $s2
      sw $t0, 0($s1)      # store result
      addi $s1, $s1,–4      # decrement pointer
      bne $s1, $zero, Loop # branch $s1!=0

ALU/branch          | Load/store          | cycle
-------------------|---------------------|-----
Loop:              |                     |     
                   | nop                 | 1   
addi $s1, $s1,-4  |                     | 2   
addu $t0, $t0, $s2| nop                 | 3   
bne $s1, $zero,   | sw $t0, 4($s1)     | 4   
Loop               |                     |     

- IPC = 5/4 = 1.25 (c.f. peak IPC = 2)
Loop Unrolling

- Replicate loop body to expose more parallelism
  - Reduces loop-control overhead
- Use different registers per replication
  - Called *register renaming*
  - Avoid loop-carried *anti-dependencies*
    - Store followed by a load of the same register
    - Aka “name dependence”
      - Reuse of a register name
Loop Unrolling Example

<table>
<thead>
<tr>
<th></th>
<th>ALU/branch</th>
<th>Load/store</th>
<th>cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop :</td>
<td>addi $s1, $s1, -16</td>
<td>lw $t0, 0($s1)</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>nop</td>
<td>lw $t1, 12($s1)</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>addu $t0, $t0, $s2</td>
<td>lw $t2, 8($s1)</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>addu $t1, $t1, $s2</td>
<td>lw $t3, 4($s1)</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>addu $t2, $t2, $s2</td>
<td>sw $t0, 16($s1)</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>addu $t3, $t4, $s2</td>
<td>sw $t1, 12($s1)</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>nop</td>
<td>sw $t2, 8($s1)</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>bne $s1, $zero, Loop</td>
<td>sw $t3, 4($s1)</td>
<td>8</td>
</tr>
</tbody>
</table>

- IPC = 14/8 = 1.75
- Closer to 2, but at cost of registers and code size
Dynamic Multiple Issue = Superscalar

- CPU decides whether to issue 0, 1, 2, ... instructions each cycle
  - Avoiding structural and data hazards

- Avoids need for compiler scheduling
  - Though it may still help
  - Code semantics ensured by the CPU
    - By stalling appropriately

- Limited benefit without compiler support
  - Adjacent instructions are often dependent
Out-of-order Execution (Dynamic Scheduling)

- Allow the CPU to execute instructions *out of order* to avoid stalls
  - But commit result to registers in order

- Example
  
  ```
  lw    $t0, 20($s2)
  add   $t1, $t0, $t2
  sub   $s4, $s4, $t3
  slti  $t5, $s4, 20
  ```

  - Can start `sub` while `add` is waiting for `lw`

- Why not just let the compiler schedule code?
Implementing Out-of-Order Execution

Basically, unroll loops in hardware:

1. Fetch instructions in program order (≤4/clock)
2. Predict branches as taken/not-taken
3. To avoid hazards on registers, *rename registers* using a set of internal registers (~80 registers)
4. Collection of renamed instructions might execute in a *window* (~60 instructions)
5. Execute instructions with ready operands in 1 of multiple *functional units* (ALUs, FPUs, Ld/St)
6. Buffer results of executed instructions until predicted branches are resolved in *reorder buffer*
7. If predicted branch correctly, *commit* results in program order
8. If predicted branch incorrectly, discard all dependent results and start with correct PC
Dynamically Scheduled CPU

Instruction fetch and decode unit

Reservation station

Reservation station

Reservation station

Reservation station

Commit unit

In-order issue

Preserves dependencies

Hold pending operands

Out-of-order execute

Results also sent to any waiting reservation stations

Can supply operands for issued instructions

Reorders buffer for register writes

Functional units

Integer

Integer

Floating point

Load-store
Takeaway points

- The 5-stage pipeline is not a bad mental model for SW developers:
  - Integer arithmetic is cheap
  - Loads can be relatively expensive
    - Especially if there is not other work to be done (e.g., linked list traversals)
    - We’ll further explain why starting on Friday
  - Branches can be relatively expensive
    - But, primarily if they are not predictable

- In addition, try to avoid long serial dependences; given double D[10]
  - Is faster than:
    - (((((D[0] + D[1]) + D[2]) + D[3]) + D[4]) + D[5]) + D[6]) + D[7])

- There is phenomenal engineering in modern processors