Today’s topics:

- Out-of-order execution
- Cache basics
An Out-of-Order Processor Implementation

- Branch prediction and instr fetch
  - R1 ← R1+R2
  - R2 ← R1+R3
  - BEQZ R2
  - R3 ← R1+R2
  - R1 ← R3+R2

- Instr Fetch Queue

- Decode & Rename
  - T1 ← R1+R2
  - T2 ← T1+R3
  - BEQZ T2
  - T4 ← T1+T2
  - T5 ← T4+T2

- Issue Queue (IQ)

- Reorder Buffer (ROB)
  - Instr 1
  - Instr 2
  - Instr 3
  - Instr 4
  - Instr 5
  - Instr 6
  - T1
  - T2
  - T3
  - T4
  - T5
  - T6

- Register File
  - R1-R32

- ALU

- Results written to ROB and tags broadcast to IQ
Example Code

<table>
<thead>
<tr>
<th>Completion times</th>
<th>with in-order</th>
<th>with ooo</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD R1, R2, R3</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>ADD R4, R1, R2</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>LW R5, 8(R4)</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>ADD R7, R6, R5</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>ADD R8, R7, R5</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>LW R9, 16(R4)</td>
<td>11</td>
<td>7</td>
</tr>
<tr>
<td>ADD R10, R6, R9</td>
<td>13</td>
<td>9</td>
</tr>
<tr>
<td>ADD R11, R10, R9</td>
<td>14</td>
<td>10</td>
</tr>
</tbody>
</table>
Cache Hierarchies

• Data and instructions are stored on DRAM chips – DRAM is a technology that has high bit density, but relatively poor latency – an access to data in memory can take as many as 300 cycles today!

• Hence, some data is stored on the processor in a structure called the cache – caches employ SRAM technology, which is faster, but has lower bit density

• Internet browsers also cache web pages – same concept
Memory Hierarchy

- As you go further, capacity and latency increase
Locality

• Why do caches work?
  ▪ Temporal locality: if you used some data recently, you will likely use it again
  ▪ Spatial locality: if you used some data recently, you will likely access its neighbors

• No hierarchy: average access time for data = 300 cycles

• 32KB 1-cycle L1 cache that has a hit rate of 95%:
  average access time = 0.95 x 1 + 0.05 x (301)
  = 16 cycles
Accessing the Cache
Accessing the Cache

Direct-mapped cache: each address maps to a unique location in cache

Byte address

101000

Offset

Data array

Sets

8-byte words

8 words: 3 index bits
The Tag Array

Direct-mapped cache: each address maps to a unique address

Byte address

Tag

Compare

Tag array

101000

Data array

8-byte words
Example Access Pattern

Direct-mapped cache: each address maps to a unique address

Assume that addresses are 8 bits long
How many of the following address requests are hits/misses?
4, 7, 10, 13, 16, 68, 73, 78, 83, 88, 4, 7, 10...

Byte address

Tag

Compare

Tag array

Data array

8-byte words

101000

Assume that addresses are 8 bits long
How many of the following address requests are hits/misses?
4, 7, 10, 13, 16, 68, 73, 78, 83, 88, 4, 7, 10…
Increasing Line Size

A large cache line size $\rightarrow$ smaller tag array, fewer misses because of spatial locality

Tag array

Byte address

Offset

Data array

10100000

Tag

32-byte cache line size or block size
**Associativity**

Set associativity $\to$ fewer conflicts; wasted power because multiple data and tags are read.
Associativity

How many offset/index/tag bits if the cache has 64 sets, each set has 64 bytes, 4 ways

Tag array

Compare

Data array

Way-1

Way-2

Byte address

10100000

Tag
Example

• 32 KB 4-way set-associative data cache array with 32 byte line sizes

• How many sets?

• How many index bits, offset bits, tag bits?

• How large is the tag array?