When and Where?

- When: Monday & Tuesday 7:00-8:50 PM
- Where: CAP 1301
- Office hours: Monday & Tuesday, after class

Go to webpage for:
- Class Slides
- Papers
- Simulator information
- Homework and project assignments
Course Description

- State of the art superscalar microprocessor design
- Emphasis on quantitative analysis
  - Homeworks and projects involve working with a high level performance simulator
- Emphasis on papers readings NOT on a textbook
  - Tutorial papers
  - Original sources and ideas papers
  - Papers covering most recent trends
Expected Background

- ECE 486/586 or equivalent
  - Basic microprocessor organization
  - Instruction sets: RISC, CISC
  - Datapath design
  - Pipelining
  - Caches
  - Basic branch prediction
- Programming experience in “C”
  - Needed for homeworks and projects
Grading Policy

- Class Participation 10%
- Homeworks (including paper reviews) 20%
- Project 30%
- Mid Term Exam 20%
- Final Exam 20%

Grading Scale:

- A: 92-100%
- A-: 86-91.5%
- B+: 80-85.5%
- B: 76-79.5%
- B-: 72-75.5%
- C+: 68-71.5%
- C: 64-67.5%
- C-: 60-63.5%
- D+: 57-59.5%
- D: 54-56.5%
- D-: 50-53.5%
- F: Below 50%
Why Study Computer Architecture

- Technology advancements require continuous optimization of cost, performance, and power
  - Moore’s law
    - Original version: Transistor scaling exponential
    - Popular version: Processor performance exponentially increasing

- Innovation needed to satisfy market trends
  - User and software requirements keep on changing
  - Software developers expecting improvements in computing power
Performance

Two important metrics

- Latency
  - Response time
    - For different hardware structures (e.g., cache access, store buffer lookup)
    - For different instructions/operations
  - Execution time from start to finish

- Throughput or bandwidth
  - Rate of task completion
  - Rate of data transfer
Instruction Cycle

- Five stages (cycles) for instruction processing:
  - Instruction fetch (IF)
  - Instruction decode, read operands (ID)
  - Execute (EX)
  - Memory read/write (MEM)
  - Write back results (WB)

- Most modern processors have many more stages
Simplified Instruction Cycle

For the remainder of this lecture, let’s simplify the instruction processing to three stages (cycles):
- Instruction Fetch (f)
- Instruction Decode and Read Operands (d)
- Execute and write results (e)
Execution Time

- Execution time (Runtime) for a program is given by:

  \[ \text{Runtime} = I \times CPI \times t_c \]

  Instructions per program
  \times \text{Cycles per instruction}
  \times \text{Time per cycle (Cycle time)}
Execution Time

- For a scalar processor (with a 3-cycle instruction processing), CPI = 3

$$\text{Runtime} = I \times 3 \times t_c$$
## Improving Performance via Basic Pipelining

\[
\text{Runtime} = I \times 1 \times t_c
\]
Superscalar Processors

- Superscalar processors: Multiple pipelines operate in parallel

<table>
<thead>
<tr>
<th>F</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>D</td>
<td>E</td>
</tr>
<tr>
<td>F</td>
<td>D</td>
<td>E</td>
</tr>
<tr>
<td>F</td>
<td>D</td>
<td>E</td>
</tr>
<tr>
<td>F</td>
<td>D</td>
<td>E</td>
</tr>
<tr>
<td>F</td>
<td>D</td>
<td>E</td>
</tr>
</tbody>
</table>

Runtime = $I \times 0.5 \times t_c$

- Superscaler techniques have been applied to both CISC and RISC processors
Superscalar Processors

- It is not guaranteed that a wide superscalar executes at maximum throughput for any given sequence of instructions
  - Instructions are not independent
    - Can’t always find more than one instruction to issue per cycle
  - Branches
    - Don’t know what instruction to fetch next
  - The processor execution resources are limited
  - Fetch and execution mechanisms
  - Cache misses
True Data Dependencies

- Also called data hazards, read-after-write (RAW) hazards
- An instruction may use a result produced by the previous instruction
  - Both instructions may not execute simultaneously in multiple pipelines
  - The second instruction must typically be stalled

\[
\begin{array}{ccc}
F & D & E \\
F & D & S & E \\
\end{array}
\]
Procedural Dependencies

- Also called control or branch hazards
- Instruction fetch implicitly depends on knowing the correct value for the program counter (PC)
  - This is (in a sense) a true dependence on the PC
  - Branches may change the program counter late in their execution, leading to pipeline stalls
Procedural Dependencies

- CISC variable length instructions introduce another procedural dependency:
  - Portions of an instruction must be decoded before the instruction length is known
Resource Conflicts

- Also called structural hazards
- If two instructions try to use the same hardware resource simultaneously, then one must wait
- Solution 1: Duplicate hardware resources
  - Can be very expensive
- Solution 2: Pipeline long latency execution units
Instruction Issue Methods

- Instruction Issue is the process of initiating instruction execution in functional units.
- Instruction Issue Policy is the mechanism the processor uses to issue instructions (and to find and examine instructions).
In-order (IO) issue and in-order (IO) execution requires instructions to be issued, executed and to complete in the same order they appear in the program.

- Simple strategy to implement BUT
- More hazards hinder performance
**IO Issue and IO Execution**

- Example: I1 requires 2 cycles to execute, I3 and I4 use same functional unit, same for I5 and I6, I5 has true dependence on I4

<table>
<thead>
<tr>
<th>Decode</th>
<th>Execute</th>
<th>Writeback</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>I1 I2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>I3 I4</td>
<td>I1 I2</td>
</tr>
<tr>
<td>3</td>
<td>I3 I4</td>
<td>I1</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>I3</td>
</tr>
<tr>
<td>5</td>
<td>I5 I6</td>
<td>I4</td>
</tr>
<tr>
<td>6</td>
<td>I6</td>
<td>I5</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>I6</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>I6</td>
</tr>
</tbody>
</table>
In-order (IO) issue and out-of-order (OO) execution allows instructions to complete in a different order.

- This prevents long operations from overly reducing performance, even for scalar processors (e.g., unrelated instructions can execute while a load from the L2 cache or a floating point divide is in progress).
### IO Issue and OO Execution

- Same example: I1 requires 2 cycles to execute, I3 and I4 use same functional unit, same for I5 and I6, I5 has true dependence on I4

<table>
<thead>
<tr>
<th>Decode</th>
<th>Execute</th>
<th>Writeback</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>I1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>I2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>I3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>I4</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>I1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>I2</td>
</tr>
<tr>
<td>4</td>
<td>I5</td>
<td></td>
</tr>
<tr>
<td></td>
<td>I6</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>I1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>I3</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>I4</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>I5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>I6</td>
</tr>
</tbody>
</table>
IO Issue and OO Execution

- If out-of-order completion is allowed, it is also possible to have an output dependence
  - Two outstanding instructions write to the same location
  - They must complete in the correct order to make sure the correct result is stored
  - This is also called a write-after-write (WAW) hazard
  
  ```
  DIV R3,R4,R5
  ...
  ADD R3,R4,R1
  ADD R5,R3,R3 ; Which R3?
  ```

- This can be overcome with register renaming
Out-of-order (OO) issue and out-of-order (OO) execution further improves performance by not stalling the processor in the presence of resource conflicts or true and output dependences

- Instructions that would cause a problem are left in an instruction window to be issued when the problem has cleared
- The processor thus can look ahead to the size of the window to find instructions to issue
**OO Issue and OO Execution**

- Same example: I1 requires 2 cycles to execute, I3 and I4 use same functional unit, same for I5 and I6, I5 has true dependence on I4

<table>
<thead>
<tr>
<th>Decode</th>
<th>Window</th>
<th>Execute</th>
<th>Writeback</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>I1, I2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>I1, I2</td>
<td>I1, I2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>I3, I4</td>
<td>I1, I3</td>
<td>I2</td>
</tr>
<tr>
<td>4</td>
<td>I4, I5, I6</td>
<td>I6, I4</td>
<td>I1, I3</td>
</tr>
<tr>
<td>5</td>
<td>I5</td>
<td>I5</td>
<td>I4, I6</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td>I5</td>
</tr>
</tbody>
</table>
OO Issue and OO Execution

- This method can cause antidependences
  - An instruction that needs to read a result may have that result overwritten by a following instruction that was issued first
  - We must make sure that the value is not overwritten until it has been read by all users
  - This is also called a write-after-read (WAR) hazard.

```
DIV R3,R4,R5
STORE A,R3
ADD R3,R4,1 ;Can’t until after store
ADD R5,R3,R3
```
Register Renaming

- Given enough on-chip storage, the hardware can automatically rename registers (as specified in the program code) to ensure that each refers to a unique location.

  - Register renaming can remove storage conflicts

    - DIV R3a,R4a,R5a
    - STORE A,R3a
    - ADD R3b,R4a,1
    - ADD R5b,R3b,R3b
Reading Assignment

- Read before Tue class
- Submit review before the beginning of Tue class:
  - Paper summary
  - Strong points (2-4 points)
  - Weak points (2-4 points)
- Review due in my email inbox before 7PM Tue
  - Plain text, no attachments
  - Subject line has to be: “ECE587_REVIEW_03_30”