# ECE 4750 Computer Architecture
# Topic 7: Advanced Processors – Out-of-Order

`http://www.csl.cornell.edu/courses/ece4750`
School of Electrical and Computer Engineering
Cornell University

revision: 2024-11-24-19-10

## List of Problems

## Problem 1. Out-of-Order Execution

In this problem, we will examine how a short assembly sequence executes on the five microarchitectures discussed in lecture. The I3L microarchitecture uses an in-order front-end, issue, writeback, and late commit. The I2OE microarchitecture uses an in-order front-end and issue with an out-of-order writeback and early commit. The I2OL microarchitecture uses an in-order front-end and issue with an out-of-order writeback and late commit. The IO2E microarchitecture uses an in-order front-end with an out-of-order issue and writeback and early commit. The IO2L microarchitecture uses an in-order front-end with an out-of-order issue and writeback and late commit.

The assembly sequence we will be using is as follows:

```
mul x1,  x2,  x3
mul x4,  x1,  x5
add x11, x12, x13
add x14, x11, x15
add x16, x14, x13
mul x6,  x7,  x8
mul x9,  x6,  x2
```

Hint: It would be probably be helpful for you to go ahead and identify all architectural RAW dependencies in this sequence. For this problem you should assume that each microarchitecture follows the details described in the lecture notes. As in lecture, all microarchitectures include an extra issue stage, a fully-pipelined four-cycle integer multiplier (stages labeled Y0, Y1, Y2, and Y3), and a single-cycle integer ALU (stages labeled either X or X0, X1, X2, X3 depending on the microarchitecture). Some microarchitectures will also have an additional commit stage (C). All pipelines are fully bypassed.

For in-order issue processors, use repeated I symbols to indicate when an instruction is stalled in the I stage waiting for a RAW hazard to be resolved. For the I2OI processor, use an *r* to indicate when an instruction is waiting in the reorder buffer. Reserve C for when the instruction actually commits. Instead of entering many duplicate *r* symbols you can use a continuous line or a series of small dots to indicate that an instruction is waiting in the queue for multiple consecutive cycles. See the pipeline diagrams in the lecture notes for more details on the proper syntax.

For the out-of-order issue processors, we use an *i* to indicate when an instruction is waiting in the issue queue and an *r* to indicate when an instruction is in the reorder buffer. Reserve C for when the instruction actually commits. Instead of entering many duplicate *i* or *r* symbols you can use a continuous line or a series of small dots to indicate that an instruction is waiting in the queue for multiple consecutive cycles. See the pipeline diagrams in the lecture notes for more details on the proper syntax.

### Part 1.A I3L: In-Order Front-End, Issue, and Writeback with Late Commit

Draw a pipeline diagram illustrating how the assembly code shown above executes on a processor with an in-order front-end, issue, and writeback with late commit. Include the microarchitectural RAW dependency arrows.

**Part 1.B I2OE: In-Order Front-End and Issue, OOO Writeback with Early Commit**

Draw a pipeline diagram illustrating how the assembly code shown above executes on a processor with an in-order front-end and issue, out-of-order writeback with early commit. Include the microarchitectural RAW dependency arrows.

**Part 1.C I2OL: In-Order Front-End and Issue, OOO Writeback with Late Commit**

Draw a pipeline diagram illustrating how the assembly code shown above executes on a processor with an in-order front-end and issue, out-of-order writeback with late commit. Include the microarchitectural RAW dependency arrows.

**Part 1.D IO2E: In-Order Front-End, OOO Issue and Writeback with Early Commit**

Draw a pipeline diagram illustrating how the assembly code shown above executes on a processor with an in-order front-end, out-of-order issue and writeback with early commit. Include the microarchitectural RAW dependency arrows.

**Part 1.E IO2L: In-Order Front-End, OOO Issue and Writeback with Late Commit**

Draw a pipeline diagram illustrating how the assembly code shown above executes on a processor with an in-order front-end, out-of-order issue and writeback with late commit. Include the microarchitectural RAW dependency arrows.

## Problem 2. Out-of-Order Scheduling

In this problem, you will be taking a closer look at what heuristic we use to dynamically schedule the instructions waiting in the issue queue for a specific short assembly sequence. The assembly sequence we will be using is as follows:

```
mul x1,  x2,  x3
mul x4,  x1,  x5
div x6,  x7,  x8
div x9,  x10, x11
div x12, x13, x14
mul x15, x12, x16
mul x17, x15, x18
```

Hint: It would be probably be helpful for you to go ahead and identify all architectural RAW dependencies in this sequence. For this problem you should assume that each microarchitecture follows the details described in the lecture notes. Assume a fully-pipelined four-cycle multiplier. Assume that all of the out-of-order issue microarchitectures use late commit. Whether they support register renaming, memory disambiguation, or branch prediction is not relevant for this question since there are no WAW/WAR hazards, memory instructions, or branches.

This problem will use a few new assumptions and terms. First, assume we have a four cycle unpipelined integer divider. This means the divider has an occupancy of four (i.e., a peak throughput of 0.25 division operations per cycle) and is a structural hazard; only one divide instruction can be using the divider at a time. Use a stage labelled as Z to indicate when an instruction is using the divider. Second, for the out-of-order issue microarchitectures, assume that all of the instructions are waiting in the issue queue. This might occur if we increase the fetch bandwidth or if there is a cache miss allowing many instructions to fill the instruction queue. As a consequence, your pipeline diagram for the out-of-order microarchitectures will not include any F or D stages. Finally, when calculating the total issue-to-commit time for an assembly sequence you should count the total number of cycles from the issue stage of the first issued instruction to the writeback stage for I3L or the commit stage for IO2L of the final instruction. This count should be inclusive. For example, the issue-to-commit time for the I3L pipeline diagram in Figure 1 would be seven cycles.

### Part 2.A Baseline I3L Microarchitecture

Draw a pipeline diagram illustrating how the assembly code shown above executes on a processor with an in-order front-end, issue, writeback/completion, and a late commit. Draw the microarchitectural dependency arrows. What is the total issue-to-commit cycle count?

| **Dynamic** | **Cycle** | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **Transaction** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** |
| 1 `mul x1, x2, x3` | F | D | I | Y0 | Y1 | Y2 | Y3 | W | |
| 2 `mul x4, x5, x6` | | F | D | I | Y0 | Y1 | Y2 | Y3 | W |

**Figure 1: Example Pipeline Diagram to Illustrate Issue-to-Commit Time**

**Part 2.B Schedule Oldest Ready Instruction First on IO2L Microarchitecture**

Draw a pipeline diagram illustrating how the assembly code shown above executes on a processor with an in-order front-end, out-of-order issue and writeback/completion with late commit. Draw the microarchitectural dependency arrows. Assume that we use the standard scheduling heuristic discussed in lecture: schedule the oldest ready instruction. What is the total issue-to-commit cycle count?

**Part 2.C Optimal Scheduling on IO2L Microarchitecture**

Determine the optimal dynamic schedule. Assume you have complete global knowledge of all of the instructions in the issue queue and how they will execute. Draw a pipeline diagram illustrating how the assembly code shown above executes using this optimal schedule. Draw the microarchitectural dependency arrows. What is the total issue-to-commit cycle count?

**Part 2.D Scheduling Comparison**

Why is the optimal schedule able to achieve higher performance as compared to the basic oldest instruction first scheduling heuristic. Do you have any thoughts on how we might be able to implement a heuristic in hardware that would be more likely to result in this optimal schedule?