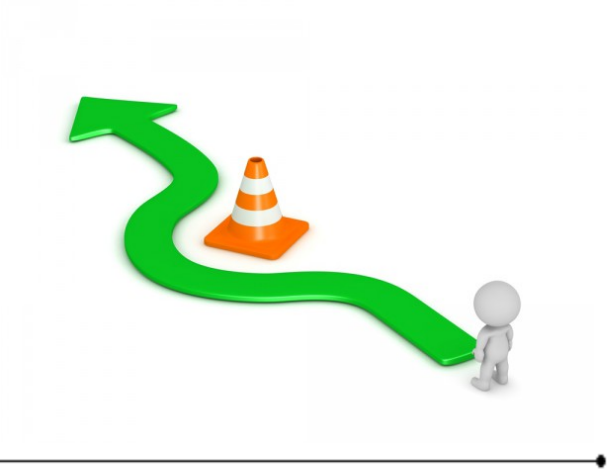
Lab Activity #2:

Pipelining – Forwarding, Hazard Detection, and Evaluation



In this week's lab, we will add support to our basic 5-stage pipeline from Lab1, to dynamically handle data and control hazards. We can then test real programs on our pipeline that have both control flow (i.e. branches) and data dependencies. We can then characterize the performance (and power/area) of our designs against our baseline OTTER (multi-cycle design from 233). As a friendly competition, we will compare groups to see who designs the processor with the best speedup and lowest energy over our benchmarks from Lab0!

Learning Objectives:

- Microarchitectural techniques for handling hazards
- Understand a basic pipelined processor microarchitecture
- Abstraction levels, including register-transfer-level modeling
- Design principles including modularity, hierarchy, and encapsulation
- Design patterns including control/datapath split and pipelined control

Introduction: *Pipelining* is an implementation technique whereby multiple instructions are overlapped in execution. Pipelining takes advantage of parallelism that exists among the actions needed to execute an instruction.

There are situations, called *hazards*, that prevent the next instruction in the instruction stream from executing during its designed clock cycle. Hazards reduce the performance from the ideal speedup gained by pipelining. There are three classes of hazards.

1. *Structural hazards* arise from resource conflicts when the hardware cannot support all possible combinations of instructions simultaneously in overlapped execution. For our pipeline, we will not have any structural hazards (e.g. our memory has two ports to handle two accesses at a time).
2. *Data hazards* arise when an instruction depends on the results of a previous instruction in a way that is exposed by the overlapping of instructions in the pipeline. These will regularly occur with our pipeline.
3. *Control hazards* arise from the pipelining of branches and other instructions that change the PC. These will also regularly occur with our pipeline.

There are three main types of data dependencies that may result in data hazards: Read After Write (RAW), Write After Read (WAR), and Write after Write (WAW). For our pipeline, the RAW hazards (or true dependencies) are the only data hazards that can occur.

To solve these hazards, we can always stall the pipeline (i.e. insert bubbles/nops) until the hazard is resolved. However, this can significantly affect performance. To solve RAW data hazards, we can use a simple hardware technique called *forwarding* (also sometimes called bypassing or short-circuiting). The key insight in forwarding is that the result of producing (or writing instruction) is not really needed by the consuming (or reading instruction) until after the producer actually produces it. If the result can be moved earlier from the pipeline register that is holding the result to where the consuming (reading) instruction needs, then stalling can be avoided.

However, even with forwarding logic, you will still need detect certain RAW hazards and stall (i.e. consider a Load instruction immediately followed by an instruction that uses the result).

For control hazards, we can also design our pipeline to stall until the branch condition and target are resolved. We can also *speculate* that the branch is not taken and continue until the branch is resolved, and if the branch was taken, squash the invalid work done in the pipeline and restart at the correct PC. In general, modern processors include more logic for dynamically predicting branches (called branch prediction). By altering what stage the branch condition and target is calculated you can also reduce the branching penalties.

Assignment:

1. Implement forwarding and hazard detection (i.e. load-use hazards) for your 5-stage pipeline implementation of the OTTER.
2. Implement logic for handling control hazards for your 5-stage pipeline.
3. Measure the performance of your OTTER for your earlier benchmarks (including using matrices of different sizes) and compare with lab0 benchmark results. Calculate the speedup for your pipelined OTTER.
4. Measure the area and power (see vivado post-synthesis reports).

Deliverables:

1. Short paragraph that provides an overview for all of your designs.
2. Answers to all questions, including spreadsheet/graphs showing your measurements.
3. HDL code for all designs