HANDLING DATA HAZARDS & CONTROL HAZARDS

Hazards: Prevent the next instruction in the instruction stream from executing during its designated clock cycle.

- Hazards reduce the performance from the ideal speedup gained by pipelining. 3 classes of hazards:
- Structural hazards: arise from resource conflicts when the hardware cannot support all possible combinations of instructions simultaneously in overlapped execution.
- 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.
- Control hazards: arise from the pipelining of branches and other instructions that change the PC.

Performance of Pipelines with Stalls

- A stall causes the pipeline performance to degrade from the ideal performance. Speedup from pipelining = [1/ (1+ pipeline stall cycles per instruction)] * Pipeline

$$\text{Speedup from pipelining} = \frac{1}{1 + \text{pipeline stall cycles per instruction}} * \text{Pipeline depth}$$

Structural Hazards

- When a processor is pipelined, the overlapped execution of instructions requires pipelining of functional units and duplication of resources to allow all possible combinations of instructions in the pipeline.
- If some combination of instructions cannot be accommodated because of resource conflicts, the processor is said to have a structural hazard.

Instances:

- When functional unit is not fully pipelined, Then a sequence of instructions using that unpipelined unit cannot proceed at the rate of one per clock cycle.
- when some resource has not been duplicated enough to allow all combinations of instructions in the pipeline to execute.

To Resolve this hazard

Stall the the pipeline for 1 clock cycle when the data memory access occurs. A stall is commonly called a pipeline bubble or just bubble, since it floats through the pipeline taking space but carrying no useful work.

Data Hazards

- A major effect of pipelining is to change the relative timing of instructions by overlapping their execution. This overlap introduces data and control hazards.
- Data hazards occur when the pipeline changes the order of read/write accesses to operands so that the order differs from the order seen by sequentially executing instructions on an unpipelined processor.

Minimizing Data Hazard Stalls by Forwarding

The problem solved with a simple hardware technique called forwarding (also called bypassing and sometimes short-circuiting).

Forwards works as:

- The ALU result from both the EX/MEM and MEM/WB pipeline registers is always fed back to the ALU inputs.
- If the forwarding hardware detects that the previous ALU operation has written the register corresponding to a source for the current ALU operation, control logic selects the forwarded result as the ALU input rather than the value read from the register file.

Data Hazards Requiring Stalls

- The load instruction has a delay or latency that cannot be eliminated by forwarding alone. Instead, we need to add hardware, called a pipeline interlock, to preserve the correct execution pattern.
- A pipeline interlock detects a hazard and stalls the pipeline until the hazard is cleared.
- This pipeline interlock introduces a stall or bubble. The CPI for the stalled instruction increases by the length of the stall.

Branch Hazards

- Control hazards can cause a greater performance loss for our MIPS pipeline . When a branch is executed, it may or may not change the PC to something other than its current value plus 4.
- If a branch changes the PC to its target address, it is a taken branch; if it falls through, it is not taken, or untaken.

Reducing Pipeline Branch Penalties

- Simplest scheme to handle branches is to freeze or flush the pipeline, holding or deleting any instructions after the branch until the branch destination is known.
- A higher-performance, and only slightly more complex, scheme is to treat every branch as not taken, simply allowing the hardware to continue as if the branch were not executed. The complexity of this scheme arises from having to know when the state might be changed by an instruction and how to “back out” such a change.
- Implemented by continuing to fetch instructions as if the branch were a normal instruction.
- The pipeline looks as if nothing out of the ordinary is happening.
- If the branch is taken, however, we need to turn the fetched instruction into a no-op and restart the fetch at the target address.

- An alternative scheme is to treat every branch as taken. As soon as the branch is decoded and the target address is computed, we assume the branch to be taken and begin fetching and executing at the target .

Performance of Branch Schemes

Pipeline speedup = Pipeline depth / [1+ Branch frequency × Branch penalty]

$$\text{Pipeline speedup} = \frac{\text{Pipeline depth}}{1 + \text{Branch frequency} \times \text{Branch penalty}}$$

The branch frequency and branch penalty can have a component from both unconditional and conditional branches.