Obstacles to Pipelining - Hazards

- Structural hazards - These occur when instructions contend for the same resources in the CPU.
- For instance, if the register file has only one write port, but for some reason the instruction stream has generated two writes to the register file in a single cycle, one of the offending pipeline stages will have to wait.
- Structural hazards can often be solved by throwing more hardware at the problem, with the penalty of increased gate count, complexity, and possibly delay.

Obstacles to Pipelining - Hazards

- Data hazards. This happens when an instruction in the pipeline depends on data from another instruction that is also in the pipeline. For instance, consider these two instructions:

```
i      add    r1, r2, r3     // r1 := r2 + r3
i+1    add    r4, r1, r5     // r4 := r1 + r5
```

- r1 is needed by instruction i+1, but the value of r1 is modified by instruction i and won't be written back to the register file before instruction i+1 reads its operands.
- There are many techniques for solving this problem. Forwarding (or bypass) is the main technique.

Obstacles to Pipelining - Hazards

Control hazards. This happens when a control-flow transfer instruction depends on results that are not ready yet

- For instance, every conditional branch presents a control flow hazard, since the condition isn't available in time to fetch the next instruction from the right place.

Dependences

Data dependence. Also called true dependence or flow dependence. This is where data needed by one instruction is produced by a previous instruction, or where data needed by one instruction flows through a chain of dependent instructions from some source.

Name dependences. This type of dependence occurs when two instructions use the same register or memory location, but there is no flow of data between the instructions. For instance, two instructions in the pipeline may both use r1 for unrelated temporary computations

Dependences

- There are two types of name dependences:

- Anti-dependence. This occurs when one instruction writes a register that will be read by another instruction. For instance:

```
add     r1, r2, r3      // r1 := r2 + r3
add     r3, r4, r5      // r3 := r1 + r4
```

- There is an anti-dependence between the two instructions because <u>the second instruction writes a register r3 that is used by the first instruction</u>.
- The processor must guarantee that the first instruction reads the correct value before the second instruction overwrites it.

Dependences

- Output dependence. This occurs when two instructions both write the same register. The processor must guarantee that the register ends up with the value from the second instruction. For instance:

```
add     r3, r2, r1      // r1 := r2 + r3
st      r3, 0(r6)       // store r3 to memory address r6
add     r3, r4, r5      // r3 := r1 + r4
```

- Here, r3 is being used for two different purposes. We must ensure that, at the end of this code, the register file entry for r3 is updated with the result from the second add instruction.

Dependences

When these dependences occur in such a way that they are exposed to the pipeline, three different types of data hazards may occur:

- RAW, or read-after-write. An instruction tries to read an operand before a previous instruction has a chance to write it. This is caused by a flow dependence.
- WAW, or write-after-write. An instruction tries to write an operand before a previous instruction has a chance to write it. Once the previous instruction writes it, the operand is left with the previous and now wrong value. This is caused by an output dependence.
- WAR, or write-after-read. An instruction writes to an operand before it can be read by a previous instruction, so the previous instruction incorrectly gets the new value. This is caused by an anti-dependence.