Pipelining (III)

Pipelining Calculation (cont.)

**Example**

- Assume there is a four-stage CPU. Each stage completes one step of the four steps of instruction execution: fetch, decode, execute, and write back. The combinational logic delay for each step is 5 ns, and the latch delay is 1 ns. Answer the following questions.

  - What is the time to execute one instruction on the 4-stage CPU?
    \[(5 + 1) * 4 = 24 \text{ ns}\]

  - Assume there is a single-stage CPU that completes all four steps in one stage. What is the time to execute one instruction on the single-stage CPU?
    \[5 * 4 + 1 = 21 \text{ ns}\]

  - What is the time to execute five instructions on the single-stage CPU?
    \[5 * 21 = 105 \text{ ns}\]

  - What is the time to execute five instructions on the four-stage CPU?
    \[(4 + 5 - 1) * (5 + 1) = 48 \text{ ns}\]

  - What is the speedup for 5 instructions using the four-stage CPU?
    \[
    \frac{105}{48} = 2.2
    \]

Real Computer Pipeline

- In real computers, the general four steps of instruction execution, Fetch, Decode, Execute, and Write-back, are often divided further to better balance the length of each step.

- MIPS pipeline
  - MIPS is a RISC developed by MIPS technologies, a classic RISC pipeline
  - Five steps to execute an instruction
    - Fetch instruction from memory
    - Read registers while decoding the instruction. The regular format of MIPS instructions allows reading and decoding to occur simultaneously.
    - Access an operand in data memory
    - Execute the operation or calculate an address
    - Write the result into a register

- Morden Pipeline
  - E.g. Pentium / PowerPC
• Seven steps to execute an instruction
  - Instruction fetch unit: get instructions
  - Decode the instructions
  - Identify where the operands are
  - Fetch the operands to the right place
  - Execute the operation
  - Store the operands
  - Operation Commit Unit: update the architectural state of the CPU

- In this course, we use the four steps for simplicity without losing the generality of this topic.

Hazards
- Pipelining can improve the throughput then speed up the instruction execution. However, there are some situations that prevent the next instruction from starting in the next cycle. We call these situations “hazards.”
- Hazards can generally be grouped into three types: structural hazards, data hazards, and control hazards.

Structural Hazards
- Structural hazards occur when a required resource is busy. For example, in dirty laundry example, if the load B starts using the washer once the load A is done with washing, the dryer is still drying the load A and is unavailable to the load B when it is done with washing.
- To address the structural hazards, more stages are usually helpful. More stages need more control hardware, but it also lowers the bound on cycle time of each stage. Modern CPU divides the four stages we discussed above into even smaller steps, so the time spent on each stage will be more balanced.
- Another way to address the structural hazards is waiting. Wait till the required resource becomes available. Actually, all hazards can be resolved by waiting. We call the waiting “stall.”

Data Hazards
- Data hazards occur in executing arithmetic or data transfer operations, when the next instruction needs to wait for the previous instruction to complete its data read/write.

- For example, if we have the following two instructions:

  LOAD R1 A # A => R1
  SUB R0 R1 R2 # R0 - R1 => R2
• R1 used by SUB won’t be ready till after executing LOAD R1 A.
• So, we have to stall one stage for the SUB instruction. The pipelining for these two instructions look like

```
load F D E W
\sub F D E W
```

• We also call this type of data hazard “load-use data hazard,” which is a specific form of data hazard, in which the date being loaded by a load instruction has not yet become available when it is needed by another instruction.

- Another way to address the data hazards is to retrieve the missing data element from internal buffers (data path) rather than waiting for it to arrive from programmer-visible register or memory. We name this solution “forwarding.”
- For example, if we have the following two instructions:

```
ADD R1 R2 R0 # R1 + R2 => R0
SUB R0 R3 R4 # R0 - R3 => R4
```

- The value of R0, which is R1 + R2, used by SUB won’t be ready till ALU finish the calculation.
- If using forwarding, the pipelining for these two instructions look like

```
add F D E W
\sub F D E W
```

- In this way, there is no need to stall one stage for the SUB instruction.

**Control Hazards**
- Control hazards occur in branch operation, where fetching next instruction depends on branch outcome.
  - Sol1: Wait until branch outcome determined before fetching next instruction.
    - Longer pipelines can’t readily determine branch outcome early. In this case, the long waiting caused by stall penalty becomes unacceptable.
  - Sol2: branch prediction

**Branch Prediction**
- A method of resolving a control hazard that assumes a given outcome for the branch and proceeds from that assumption rather than waiting to a certain actual outcome.
  - Predict outcome of branch, only stall if prediction is wrong.
- Prediction scheme:
  - Assume the branch is not taken:
  - Continue executing the sequential instructions. If the branch is taken, the instruction that are being fetched and decoded must be discarded. Execution continues at the branch target.
  - Discarding instructions means flush these instructions in the pipeline.