**PIPELINING**

implementation technique in which multiple instructions are overlapped in execution

As long as we have separate resources for each stage, we can pipeline the tasks.

MIPS instructions take five steps:
1. Fetch instruction from memory
2. read registers while decoding the instruction
3. execute the operation or calculate an address
4. access an operand in data memory or write register (R-type)
5. write the result into a register (LW)

**IDEAL CASE:**
- balanced stages
- no overhead

\[ \text{SPEED UP:} \]

\[ \text{Time between instructions} = \frac{\text{Time between instructions}_{SP}}{\text{Number of pipe stages}} \]
PIELINING improves performance by increasing instruction throughput as opposed to decreasing execution time of an individual instruction.

Real programs execute billions of instructions. Throughput is the metric that matters!!

**PIPELINE HAZARDS** situations when the next instruction cannot execute in the following cycle.

**TYPE OF HAZARDS:**
- Structural
- Control
- Data

**STRUCTURAL HAZARDS** hardware cannot support the combination of instructions that we want to execute in the same clock cycle.

Example: suppose we have a single memory for instructions and data.
CONTROL HAZARDS

arising from the need to make a decision based on the result of one instruction while others are executing

**Example**

```
add $4, $5, $6
beq $1, $2, 40
lw $3, 300($0)
```

I do not know for sure what instruction should be fetched! It depends if the branch is taken or not.
control hazard solution
  * stall (= bubble)
  * predict
  * delayed decision

\[
\begin{align*}
\text{add} & \quad \$4, \$5, \$6 & \quad \text{IF} & \quad \text{ID} & \quad \text{EX} & \quad M & \quad \text{WB} \\
\text{beq} & \quad \$1, \$2, \$40 & \quad \text{IF} & \quad \text{ID} & \quad \text{EX} & \quad \text{M} & \quad \text{WB} \\
\text{lw} & \quad \$3, \$300(\$0) & \quad \text{IF} & \quad \text{ID} & \quad \text{EX} & \quad M & \quad \text{WB} \\
\end{align*}
\]

→

\text{BRANCH NOT TAKEN}

\[
\begin{align*}
\text{add} & \quad \$4, \$5, \$6 & \quad \text{IF} & \quad \text{ID} & \quad \text{EX} & \quad M & \quad \text{WB} \\
\text{beq} & \quad \$1, \$2, \$40 & \quad \text{IF} & \quad \text{ID} & \quad \text{EX} & \quad M & \quad \text{WB} \\
\end{align*}
\]

\[\rightarrow \text{or} \quad \$7, \$8, \$9\]

\[\uparrow \]

\text{BRANCH TAKEN}
DELAYED DECISION:
The delayed branch always executes the next instruction with the branch taking place after that one instruction delay.

Software will place an instruction immediately after the delayed branch instruction that is not affected by the branch.

```
beq $1, $2, 40
add $4, $5, $6 ← delayed branch slot
lw $3, 300($0)
```

DATA HAZARDS
An instruction depends on the results of a previous instruction still in the pipeline.

Solution:
Getting the missing item early from the internal resources (FORWARDING = BYPASSING)

Example
```
add $s0, $t0, $t1
sub $t2, $s0, $t3
```
Forwarding works well but cannot prevent all pipeline stalls!!

Load-use data hazard

\[ \text{lw } $s0, 20($t1) \]
\[ \text{sub } $t2, $s0, $t3 \]

Delayed loads – solution to load-use data hazard

Software follows a load with an instruction independent of that load