So, ideally, pipeline looks like:

1. Instruction decode & register read
2. Execute operation or calculate address
3. Write result to register
4. Memory access
5. Write back to register

Pipeline Stages (Carly Myers):

- Motivation
- 1. Break instructions execution into
- multiple stages (e.g., ID, EX, MEM, WT)
- As the instruction finishes a stage
- of an instruction, issue a new instruction, so the
- next stage of the next instruction.

Chapter 6, P44 (just 6.1, 6.5, 6.6, 6.9)
Thus, instead of the total execution being:

\[ T = N \times CPI \]

where \( N = \text{total number of instructions executed} \) and \( CPI = \text{cycles per instruction} \), the total pipelined execution time is (ideally)

\[ T = N + CPI - 1 \]

Typically, each stage in the pipeline is 1 cycle, so CPI is the number of stages.
"Ideally" an instruction can start executing every cycle, but that isn't always possible.

Why?
- Structural hazards
- Control hazards
- Data hazards

Structural Hazards: When two instructions in different stages of the pipeline require the same hardware resource.

- Example: If a single memory is used (with a single)
  then IF and Mem stages cannot occur simultaneously.
  - See last instruction in previous drawing
  - Other structural hazards might include bus contention, contention for ALU
  - Can be aided by good choice of ISA.

- When a hazard occurs, an instruction may be delayed until the resource is free
  - This results in a stage being "untitled" for a cycle causing a bubble.
Control Hazards

What instruction should follow a branch instruction?
- Don't know until the condition is computed (target address can be computed at same time).

Solutions:
1. Stall the pipeline
   - i.e. don't perform IF for the next instruction until the branch has been resolved (i.e. to branch or not).
   - this inserts bubbles into the first stage

2. Perform branch prediction
   - guess which way the branch will go and start IF for that instruction on time
   - if wrong, flush the incorrect instruction from the pipeline

- it's important to guess right most of the time:
  - possibilities: always guess branch not taken
  - use static/compile analysis (e.g. branch back usually at bottom of loop)
  - use branch prediction hardware
(Solutions to control hazards continued)

3. Always execute the instruction following the branch, regardless of whether the branch is taken or not - "branch delay slot"

- Compiler or assembler makes sure the instruction following the branch is the appropriate function

- E.g. move an instruction from above the branch into the branch delay slot.

- Not always possible (usually about 50% of the time)

Data Hazards

What if the value required by one instruction in the pipeline is being computed by another instruction in the pipeline?

Example:  
\[
\text{add } \$50, \$51, \$52 \\
\text{sub } \$54, \$53, \$30
\]

Solutions: 1) Don't have the compiler schedule these instructions like this - can't always avoid.
2) Stall the second instruction until \$50 has been written to.
(Solution to data hazards, continued)

3. Perform "forwarding" (aka "bypassing")
   - write directly from one stage of the pipeline to another
   - bypassing the need to stall for a register write:

Example from before:

```
add $50, $51, $52
sub $54, $53, $50
```

```
IF  ID    ALU  MEM  REGW
IF  ID    ALU  MEM  REGW
```

You still have to be careful to ensure that the
"source" stage executes before the "target" stage
during bypassing.
- still might need to stall

Example

```
lw $51, 100($52)
add $53, $51, $52
```

```
IF  ID  ADDRESS  READ  MEM  REGW
IF  ID  BUBBLE  ADD  ALU  MEM  RW
```

Compiler can help:

```
lw $51, 100($52)
lw $52, 200($53)
5sw $52, 100($54)
5sw $51, 200($53)
```

3. interchange these two
   to remove data hazard.