Pipelined MIPS

While a typical instruction takes 3-4 cycles (i.e. 3-4 CPI), a pipelined processor targets 1 CPI (and gets close to it).

Break datapath into 5 stages

- Each stage has its own functional units.
- Each stage can execute in 2ns
  - Just like the multi-cycle implementation

It shows the rough division of responsibilities.

The buffers between stages are not shown.
Problem 1. How can the same adder perform IF and EX in cycle 3? We need an extra adder! Gradually we need to modify the data path for the multi-cycle implementation.

Problem 2. How can we read instruction memory and data memory in the same clock cycle? We had to return to Harvard architecture!
Uniformity is simplicity

- **Enforce uniformity**
  - Make all instructions take 5 cycles.
  - Make them have the same stages, in the same order
    - Some stages will do nothing for some instructions

```
R-type         IF  ID  EX  NOP  WB

Clock cycle

   1  2  3  4  5  6  7  8  9
add Ssp, $sp, -4 IF  ID  EX  NOP  WB
sub $v0, $a0, $a1 IF  ID  EX  NOP  WB
lw  St0, 4($sp)   IF  ID  EX  MEM  WB
or  $s0, $s1, $s2 IF  ID  EX  NOP  WB
lw  St1, 8($sp)   IF  ID  EX  MEM  WB
```

- Stores and Branches have NOP stages, too…

```
store           IF  ID  EX  MEM  NOP
branch          IF  ID  EX  NOP  NOP
```
Speedup

The steady state throughput is determined by the time $t$ needed by one stage.

The length of the pipeline determines the pipeline filling time

If there are $k$ stages, and each stage takes $t$ time units, then the time needed to execute $N$ instructions is

$$k.t + (N-1).t$$

Estimate the speedup when $N=5000$ and $k=5$
Hazards in a pipeline

Hazards refer to conflicts in the execution of a pipeline. One example is the need for the same resource (like the same adder) in two concurrent actions. This is called structural hazard. To avoid it, we have to replicate resources. Here is another example:

\[
\begin{align*}
\text{lw } & \text{s1, 4(sp)  IF ID EX MEM, WB} \\
\text{add } & \text{s0, s1, s2  IF ID EX MEM WB}
\end{align*}
\]

Notice the second instruction tries to read \(s1\) before the first instruction complete the load! This is known as data hazard.
Avoiding data hazards

One solution is in insert bubbles (means delaying certain operation in the pipeline)

lw $s1, 4($sp)  IF ID EX MEM WB
add $s0, $s1, $s2  IF nop nop nop ID

Another solution may require some modification in the datapath, which will raise the hardware cost

Hazards slow down the instruction execution speed.
There is no guarantee! The next instruction has to wait until the predicate ($s1=0) is resolved. Look at the tasks performed in the five steps – the predicate is evaluated in the EX step. Until then, the control unit will insert **nop (also called bubbles)** in the pipeline.
The Five Cycles of MIPS

(Instruction Fetch)

\[ \text{IR} := \text{Memory}[\text{PC}]; \text{PC} := \text{PC} + 4 \]

(Instruction decode and Register fetch)

\[ \text{A} := \text{Reg}[\text{IR}[25:21]], \text{B} := \text{Reg}[\text{IR}[20:16]] \]
\[ \text{ALUout} := \text{PC} + \text{sign-extend(IR}[15:0]) \]

(Execute|Memory address|Branch completion)

Memory refer: \[ \text{ALUout} := \text{A} + \text{IR}[15:0] \]
R-type (ALU): \[ \text{ALUout} := \text{A op B} \]
Branch: if \text{A} = \text{B} then \text{PC} := \text{ALUout}

(Memory access | R-type completion)

LW: \[ \text{MDR} := \text{Memory}[\text{ALUout}] \]
SW: \[ \text{Memory}[\text{ALUout}] := \text{B} \]
R-type: \[ \text{Reg}[\text{IR}[15:11]] := \text{ALUout} \]

(Write back)

LW: \[ \text{Reg}[[20:16]] := \text{MDR} \]
sub $s1, $t1, $t2
beq $s1, $zero L
Some instruction here

An alternative approach to deal with this is for the compiler (or the assembler) to insert NOP instructions, or reorder the instructions.