9 - Pipelining

9.1 Introduction

Pipelining doesn't help latency of single task:

- It helps the throughput of the entire wordload

Multiple tasks operating simultaneously using different resources

Possible delays:

- Pipeline rate limited by slowest pipeline stage
- Stall of dependencies

9.2 MIPS Pipeline Stages

Five execution stages:

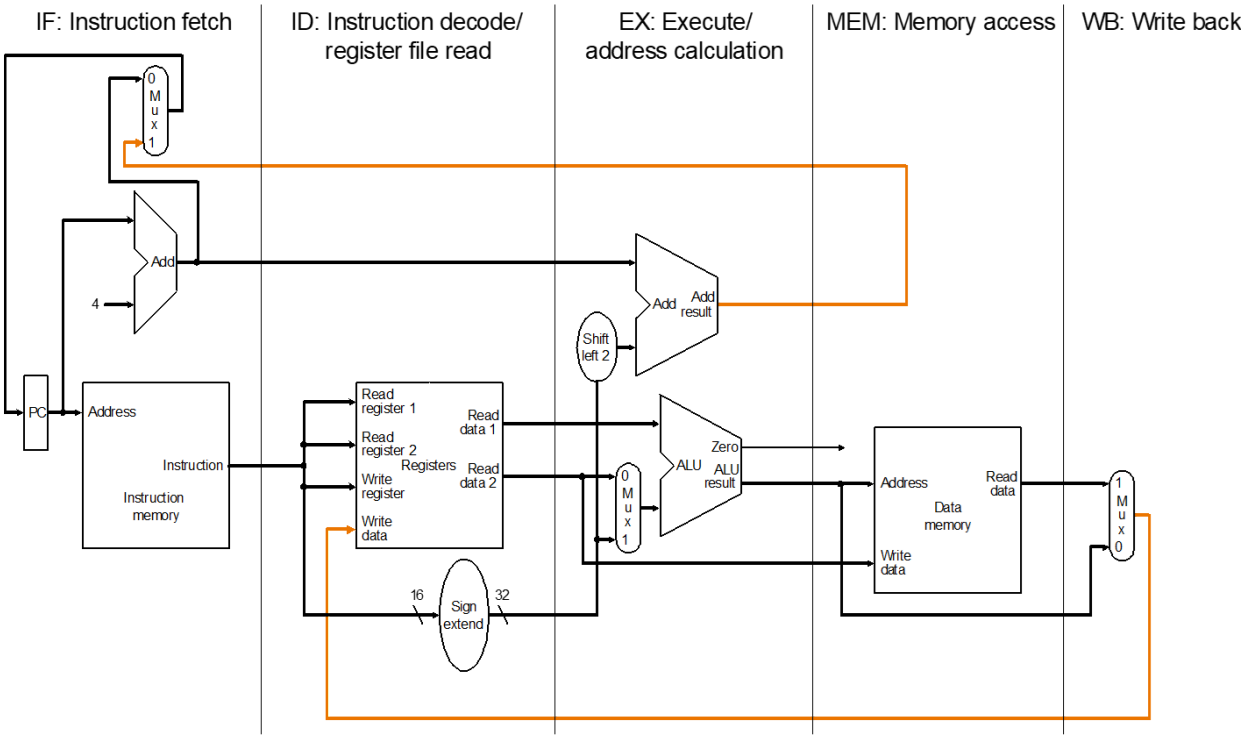
1. `IF` : Instruction Fetch
2. `ID` : Instruction Decode and Register Read
3. `EX` : Execute an operation or calculate an address
4. `MEM` : Access an operand in data memory
5. `WB` : Write Back the result into a register

Idea:

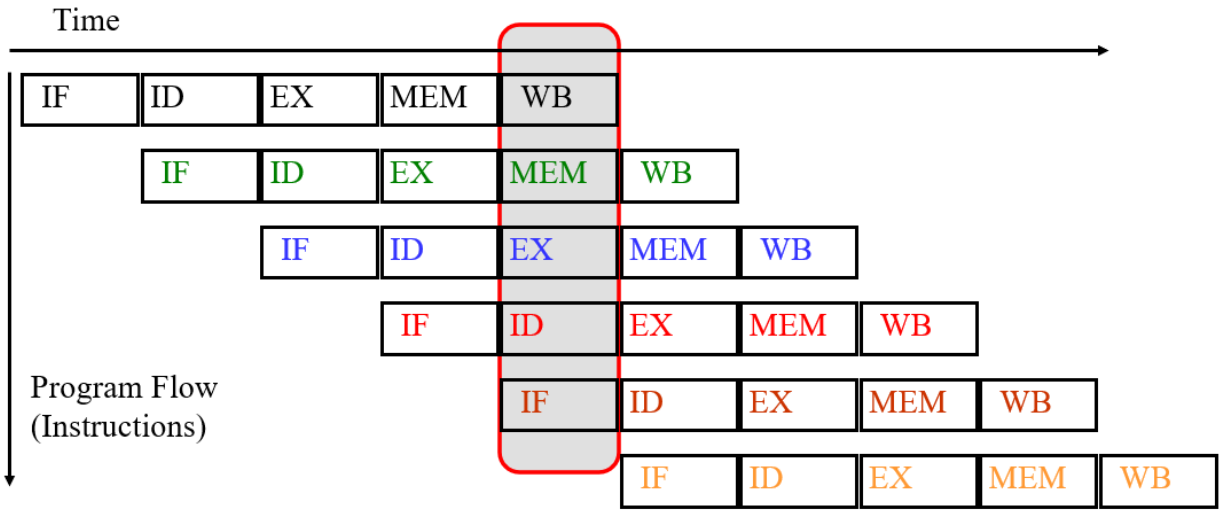
- Each execution stage takes 1 clock cycle
- General flow of data is from one stage to the next

Exceptions:

- Update of PC and write back of register file



9.2.1 Pipelined Execution: Illustration



9.3 Pipeline Datapath

Single-cycle implementation:

- Upload all state elements (PC, register file, data memory) at the end of a clock cycle

Pipelined implementation:

- One cycle per pipeline stage
- Data required for each stage needs to be stored separately

单周期实现：

在单周期处理器中，每个指令从开始到结束都在一个时钟周期内完成。因此，所有的状态元素（如程序计数器（PC）、寄存器文件、数据内存等）都是在时钟周期的末尾更新的。这意味着，在下一个周期开始之前，CPU执行完整条指令的所有步骤。这种方法简单、清晰，但速度受限于最慢的指令，因为所有指令都必须在同一个周期长度内完成。

流水线实现：

与单周期不同，流水线处理器将指令执行划分为几个阶段，每个阶段在一个时钟周期内完成一部分任务。这样做的目的是让不同指令的不同部分能够并行执行，从而在给定时间内完成更多的指令。

现在，关于为什么每个阶段需要的数据需要分别存储，这里有几点关键原因：

1. **防止数据冲突和冒险**：在流水线中，多条指令会重叠执行。一条指令的某个阶段可能需要使用前一条指令的结果。如果所有数据都存储在同一位置，一条指令的输出可能会覆盖另一条指令的关键数据，导致错误。通过在每个阶段结束时存储数据，我们可以保证每条指令都能访问其所需的正确数据，而不会被其他同时执行的指令干扰。
2. **阶段间同步**：由于每个阶段都在自己的时钟周期内独立操作，所以必须有一种机制确保数据在正确的时间传输到下一个阶段。这意味着每个阶段结束时，其输出数据必须被存储在一个地方，以便下一个阶段在下一个时钟周期开始时可以使用。这通常是通过在各个阶段之间使用寄存器（被称为流水线寄存器）来实现的。
3. **增强处理能力**：将每个阶段需要的数据分开存储，意味着当一个阶段正在处理一条指令时，其他阶段可以同时读取和处理来自/要传递到其他指令的数据。这消除了处理过程中的闲置时间，允许处理器更快地执行指令序列。

简而言之，流水线实现通过在每个阶段的结尾单独存储状态和数据，使得多条指令可以同时且高效地在不同阶段执行，增加了整个处理器的吞吐量和效率。而这种分阶段存储的需求来源于并行指令执行过程中对数据完整性和正确同步的基本需求

Data used by subsequent instructions:

- Store in programmer-visible state elements: `PC`, register file and memory

Data used by same instruction in later pipeline stages:

- Additional registers in datapath called pipeline registers
- `IF/ID`: register between `IF` and `ID`
- `ID/EX`: register between `ID` and `EX`
- `EX/MEM`: register between `EX` and `MEM`
- `MEM/WB`: register between `MEM` and `WB`

在流水线处理器架构中，如何管理和存储在各个阶段中产生并且在后续阶段中需要使用的数据。为了保持处理器的高效运行，防止数据冲突和数据冒险，流水线架构使用了特殊的寄存器，称为流水线寄存器。下面详细解释这段话的内容。

程序可见状态元素：

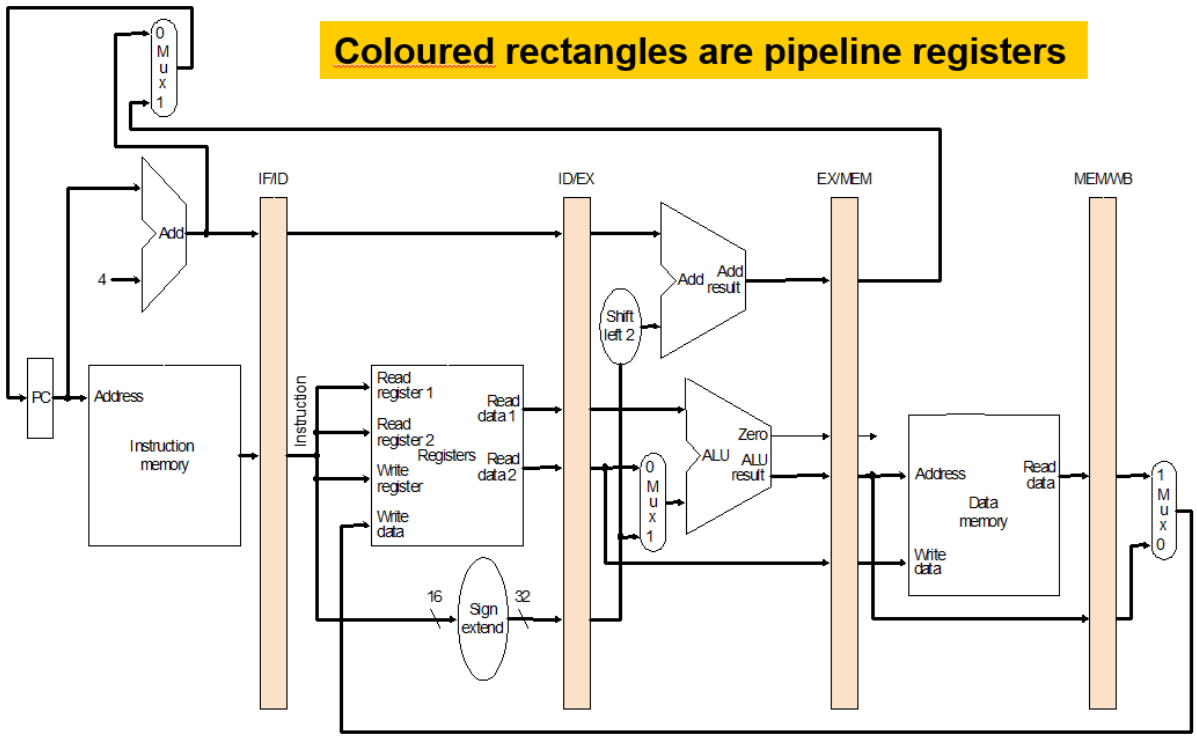
- **程序计数器（PC）**：它存储的是下一条要执行的指令的地址。
- **寄存器文件**：这是一组寄存器，用于存储在指令执行过程中需要的数据。
- **内存**：这是一个更大的数据存储区域，用于存储指令以及指令操作的数据。

这些元素对程序员是可见的，因为他们在编写程序时可以直接或间接地操作这些元素。

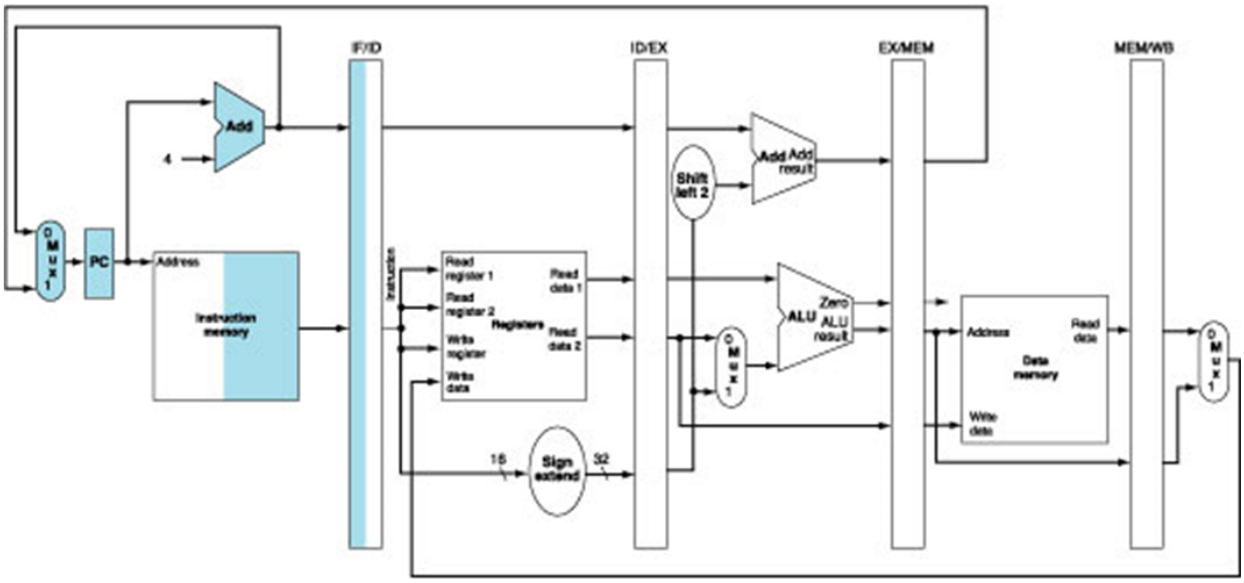
流水线寄存器：

流水线寄存器是在流水线的各个阶段之间放置的寄存器。它们在每个时钟周期结束时捕获并存储当前阶段的输出，这些输出数据将在下一个时钟周期中的下一个阶段被使用。这样做是为了保证即使在下一个时钟周期中当前阶段有新的数据产生，也不会影响下一阶段需要的数据。

- **IF/ID** ：这个寄存器存储的是从“指令取指”阶段（IF）到“指令译码”阶段（ID）的数据。这包括被取出的指令以及程序计数器（PC）的值。
- **ID/EX** ：这个寄存器在“指令译码”阶段（ID）和“执行”阶段（EX）之间。它存储的数据通常包括译码后的指令信息，操作数，以及要执行的下一操作的必要信息。
- **EX/MEM** ：这个寄存器连接“执行”阶段（EX）和“访问内存”阶段（MEM）。它会存储ALU的操作结果，以及对内存的任何访问指令（如果有的话）。
- **MEM/WB** ：这个寄存器位于“访问内存”阶段（MEM）和“写回”阶段（WB）之间。它会存储从内存读取的数据（如果指令涉及内存读取的话）和/或ALU的结果，这些数据需要写回到寄存器文件中。



9.3.1 IF Stage

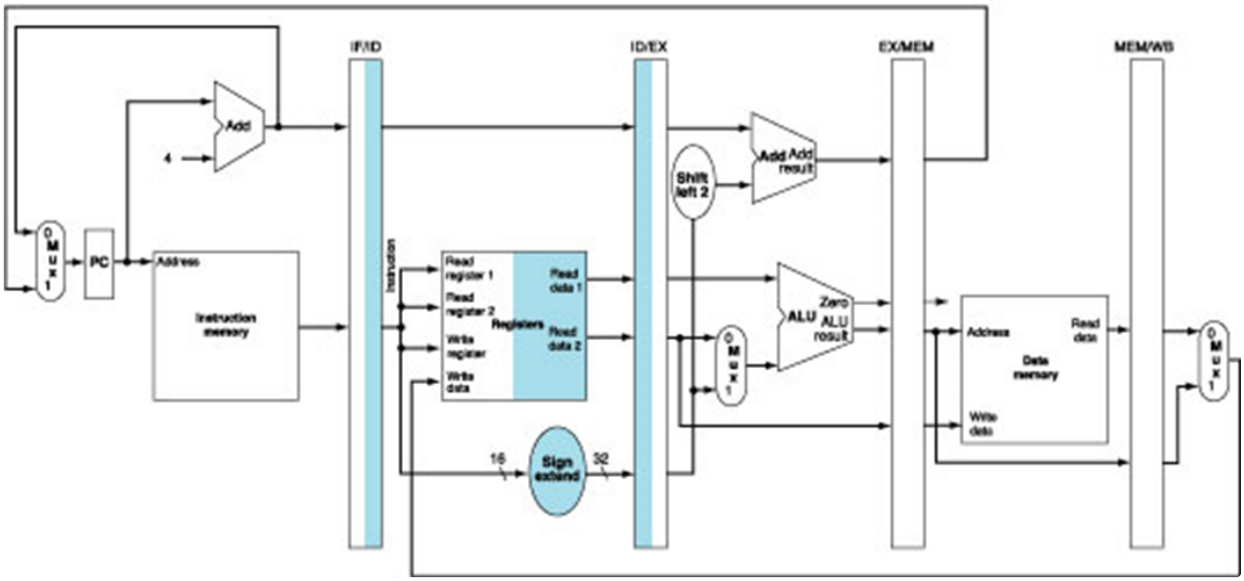


At the end of a cycle, IF/ID receives (stores):

- Instruction read from InstructionMemory[PC]
- PC+4

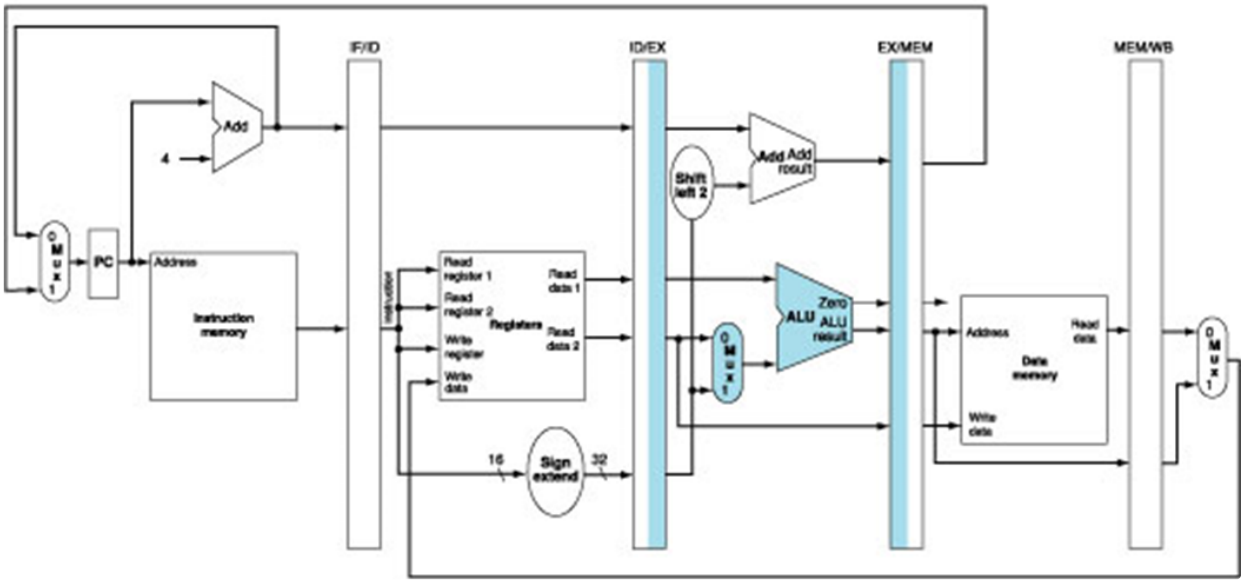
PC+4 also connected to one of the MUX's inputs

9.3.2 ID Stage



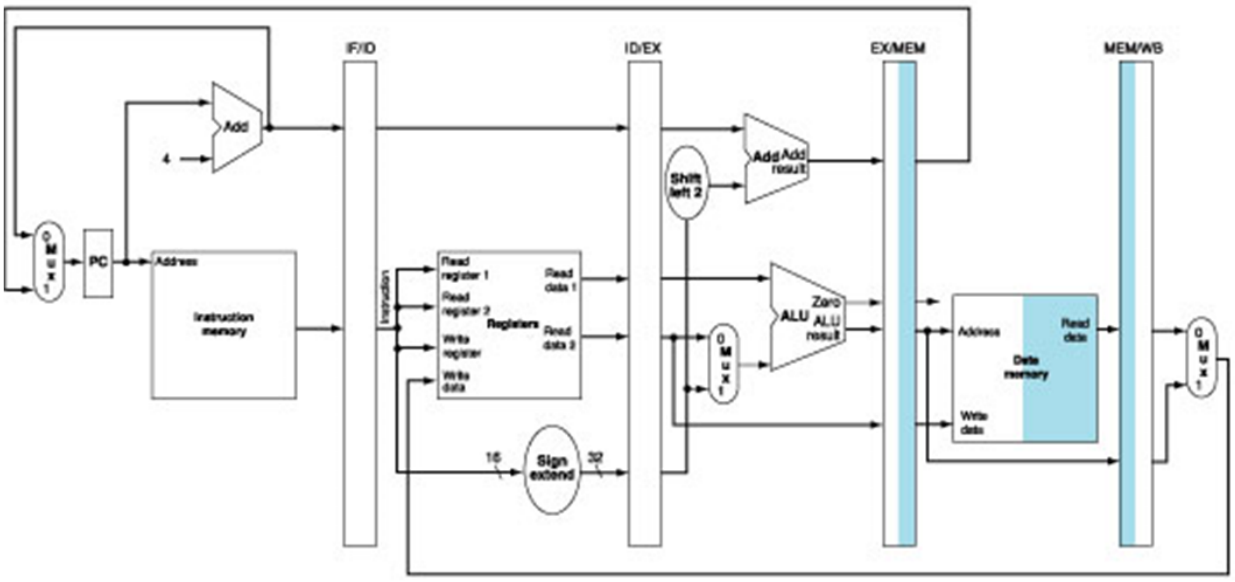
At the beginning of a cycle IF/ID register supplies:	At the end of a cycle ID/EX receives:
❖ Register numbers for reading two registers ❖ 16-bit offset to be sign-extended to 32-bit	❖ Data values read from register file ❖ 32-bit immediate value ❖ **PC + 4**

9.3.3 EX Stage



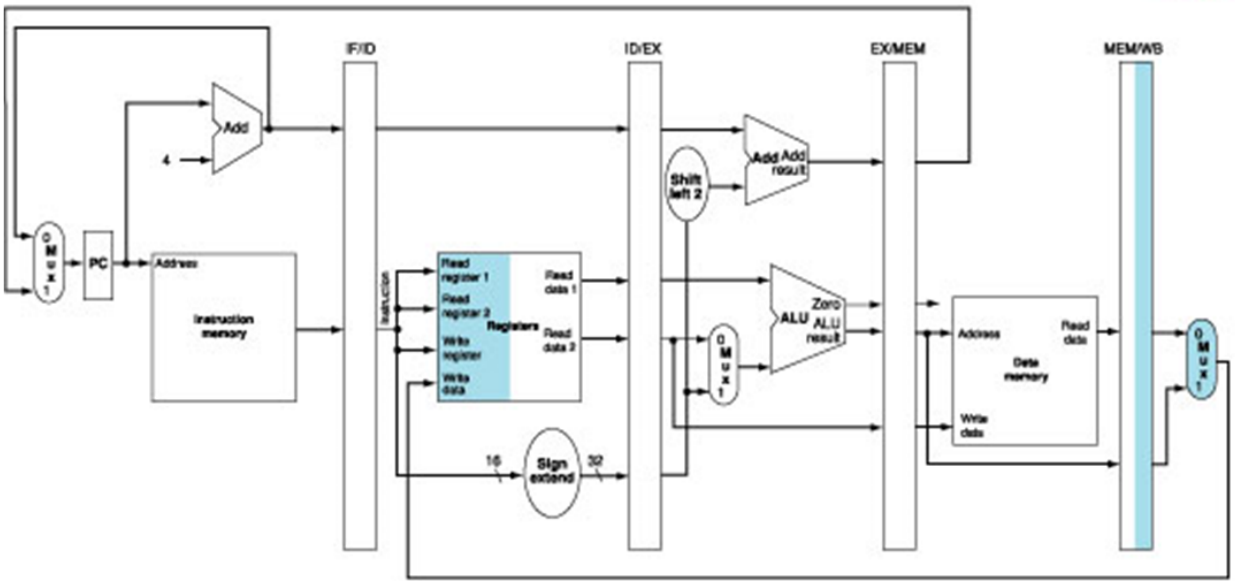
At the beginning of a cycle ID/EX register supplies:	At the end of a cycle EX/MEM receives:
❖ Data values read from register file ❖ 32-bit immediate value ❖ **PC + 4**	❖ (PC + 4) + (Immediate x 4) ❖ ALU result ❖ **isZero?** signal ❖ Data Read 2 from register file

9.3.4 MEM Stage



At the beginning of a cycle EX/MEM register supplies:	At the end of a cycle MEM/WB receives:
❖ (PC + 4) + (Immediate x 4) ❖ ALU result ❖ `isZero?` signal ❖ Data Read 2 from register file	❖ ALU result ❖ Memory read data

9.3.5 WB Stage



At the beginning of a cycle MEM/WB register supplies:	At the end of a cycle
❖ ALU result ❖ Memory read data	❖ Result is written back to register file (if applicable) ❖ **There is a bug here…….**

1. **取指阶段（IF - Instruction Fetch）：**
   - **提供的数据：** 这个阶段不接收来自上一个流水线寄存器的数据，而是从程序计数器（PC）获取当前指令的地址。
   - **接收的数据（进入 IF/ID 寄存器）：** 该阶段从内存中获取指令，并将指令本身以及下一条指令的地址（通常是当前 PC + 4）提供给 IF/ID 寄存器。
2. **指令译码/寄存器读取阶段（ID - Instruction Decode）：**
   - **提供的数据（来自 IF/ID 寄存器）：** 该阶段接收上一阶段取得的指令和下一条指令的地址。
   - **接收的数据（进入 ID/EX 寄存器）：** 该阶段解码指令，读取必要的寄存器，并将以下信息传递给下一阶段：解码的操作码和操作数、从寄存器文件中读取的数据、即将执行的指令的控制信号（例如，ALU应执行的操作，是否需要访问内存等）。
3. **执行/地址计算阶段（EX - Execution）：**
   - **提供的数据（来自 ID/EX 寄存器）：** 此阶段接收解码的指令数据、操作数、以及控制信号。

- **接收的数据（进入 EX/MEM 寄存器）：** ALU在这里执行计算或地址计算。该阶段将计算结果、任何要写入的值（对于存储指令）、计算出的内存地址（如果适用）以及控制信号传递给下一阶段。

4. **内存访问阶段（MEM - Memory Access）：**

- **提供的数据（来自 EX/MEM 寄存器）：** 这个阶段接收来自ALU的计算结果，内存地址，要写入的值，以及控制信号。
- **接收的数据（进入 MEM/WB 寄存器）：** 根据指令的需要，可能会从内存读取数据或向内存写入数据。它将读取的数据（如果有）、之前ALU的计算结果、以及控制信号传递到下一阶段。

5. **写回阶段（WB - Write-Back）：**

- **提供的数据（来自 MEM/WB 寄存器）：** 此阶段接收可能从内存中读取的数据、ALU的计算结果，以及控制信号，确定是否需要将结果写回寄存器文件。
- **接收的数据：** 在此阶段，数据被写回到寄存器文件中，完成指令的执行。由于这是流水线的最后一个阶段，因此不需要再将数据传递给另一个流水线寄存器。

9.3.6 Corrected Datapath

Observe the “Write register” number

- Supplied by the `IF/ID` pipeline register
- It is NOT the correct write register for the instruction now in `WB` stage

Solution:

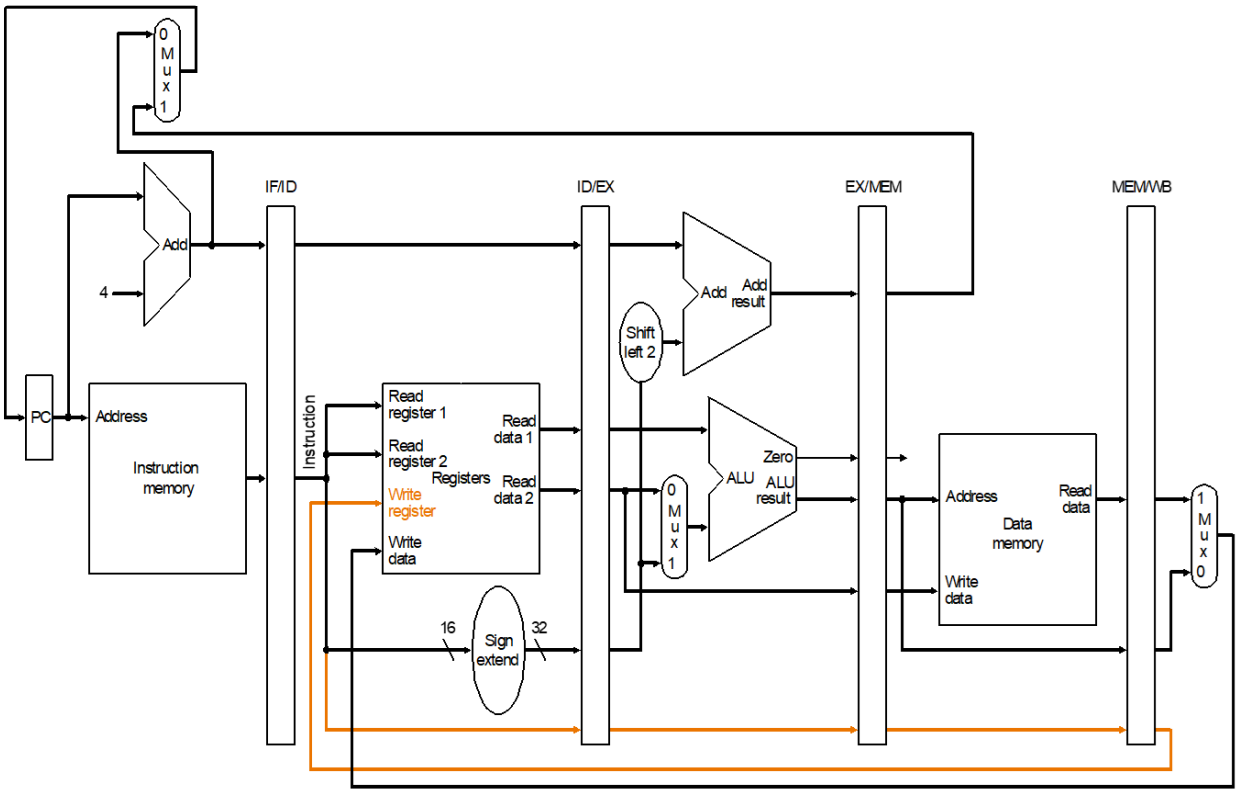
- Pass “Write register” number from `ID/EX` through `EX/MEM` to `MEM/WB` pipeline register for use in `WB` stage
- i.e. let the “Write register” number follows the instruction through the pipeline until it is needed in `WB` stage

在流水线的设计中，每条指令通过各个阶段时，必须保持其相关数据的可用性，直到这些数据真正需要为止。观察到的问题是，“写寄存器”号（即将要写入的目标寄存器的地址）在流水线的某个点上不可用或不正确。在这种情况下，指令在“写回”（WB）阶段，需要知道数据应写回哪个寄存器，但是由于在流水线的早期阶段（IF/ID阶段）提供的“写寄存器”号，并没有随着指令一起传递，所以到达WB阶段时，它不是正确的寄存器号。

解决方案：

1. **传递“写寄存器”号码：** 解决方法是，从“指令译码”（ID）阶段开始，让“写寄存器”号随着指令一起通过流水线，经过“执行”（EX）和“内存访问”（MEM）阶段，最终到达“写回”（WB）阶段。这意味着在ID阶段确定的“写寄存器”号需要被存储并传递到流水线的后续阶段。
2. **通过流水线寄存器：** 为了实现这一点，系统引入了流水线寄存器，这些寄存器位于各个阶段之间。特别地，从ID/EX到EX/MEM，再到MEM/WB的流水线寄存器将携带这个“写寄存器”号。这保证了当指令到达WB阶段时，系统知道应该将数据写回哪个寄存器。

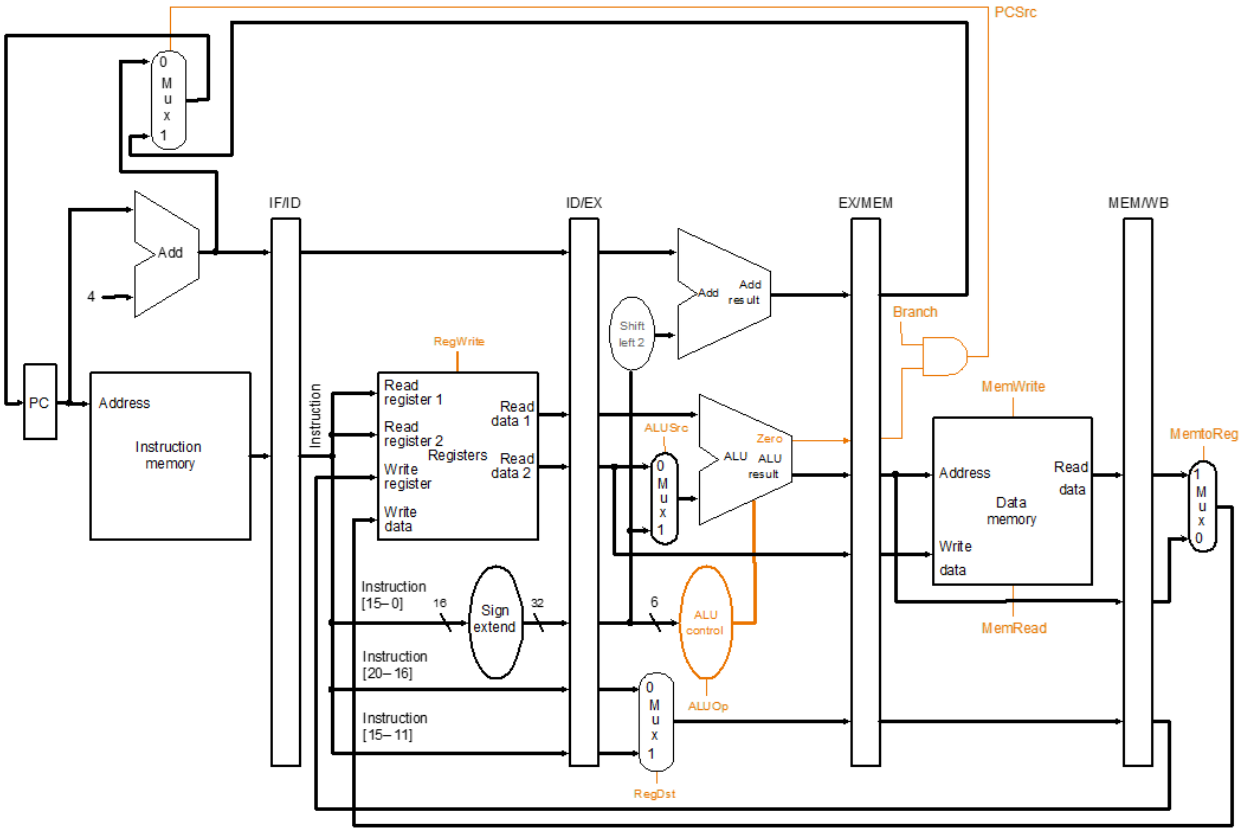
总的来说，这种更正确保了数据的准确性，并使流水线在处理每条指令时能够维持正确的状态。这是流水线设计中处理各种依赖关系的通用方法之一，确保数据和控制信息能够在需要时可用，从而避免错误和性能下降。



9.4 Pipeline Control

Main Idea

- Same control signals as single-cycle datapath
- Difference: Each control signal belongs to a particular pipeline stage



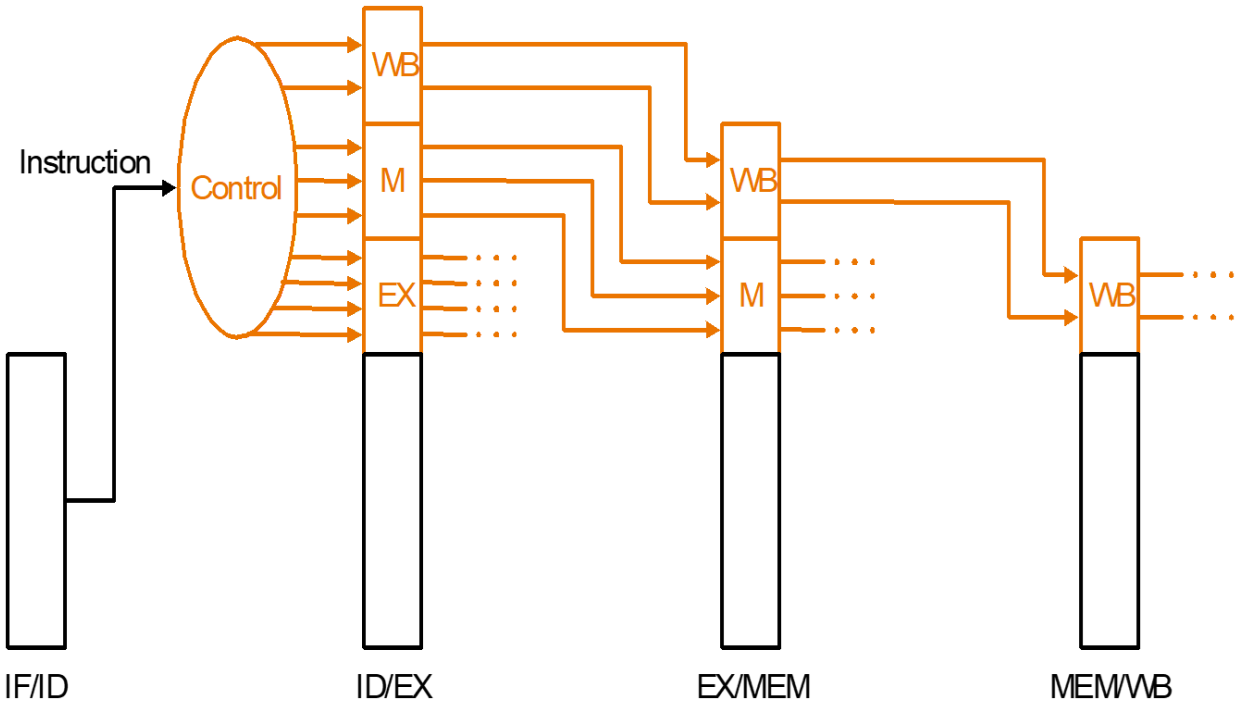
9.4.1 Grouping

Group control signals according to pipeline stage

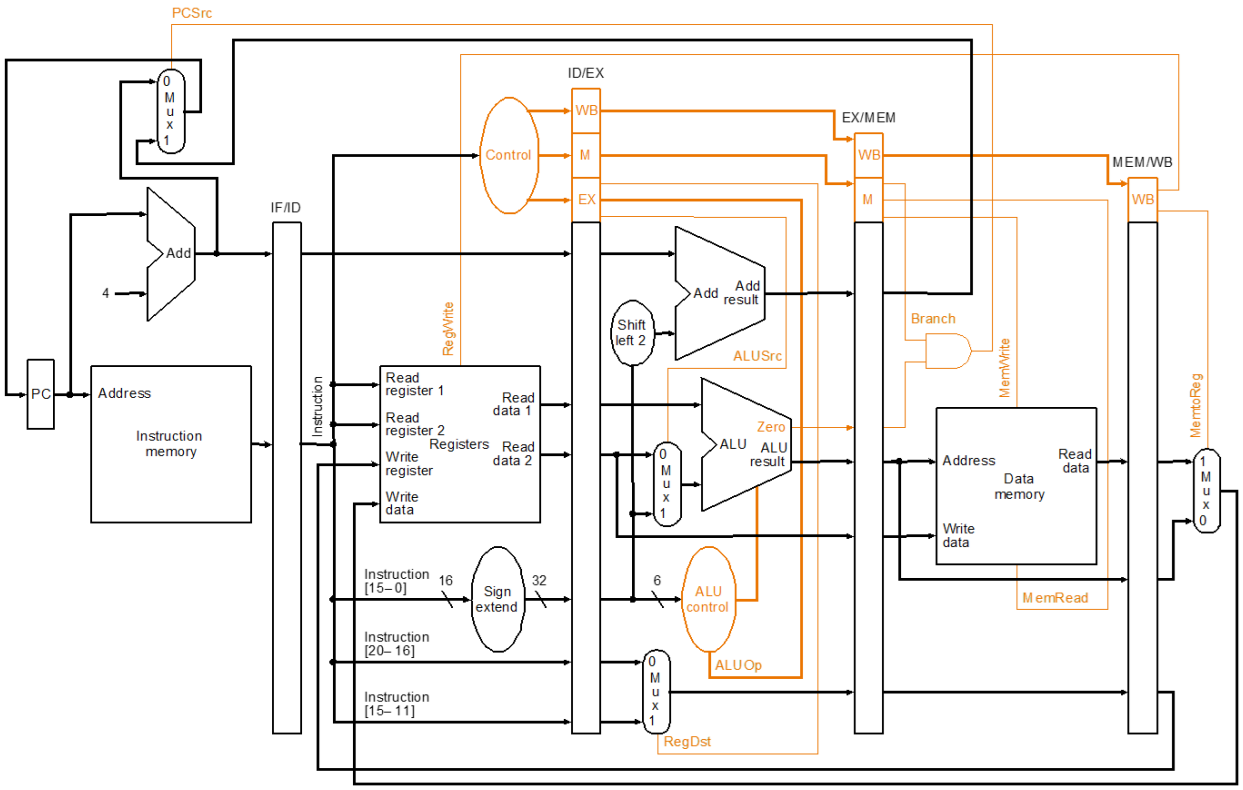
	RegDst	ALUSrc	MemTo Reg	Reg Write	Mem Read	Mem Write	Branch	ALUop	
								op1	op0
R-type	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

	EX Stage				MEM Stage			WB Stage	
	RegDst	ALUSrc	ALUop		Mem Read	Mem Write	Branch	MemTo Reg	Reg Write
			op1	op0					
R-type	1	0	1	0	0	0	0	0	1
lw	0	1	0	0	1	0	0	1	1
sw	X	1	0	0	0	1	0	X	0
beq	X	0	0	1	0	0	1	X	0

9.4.2 Implementation

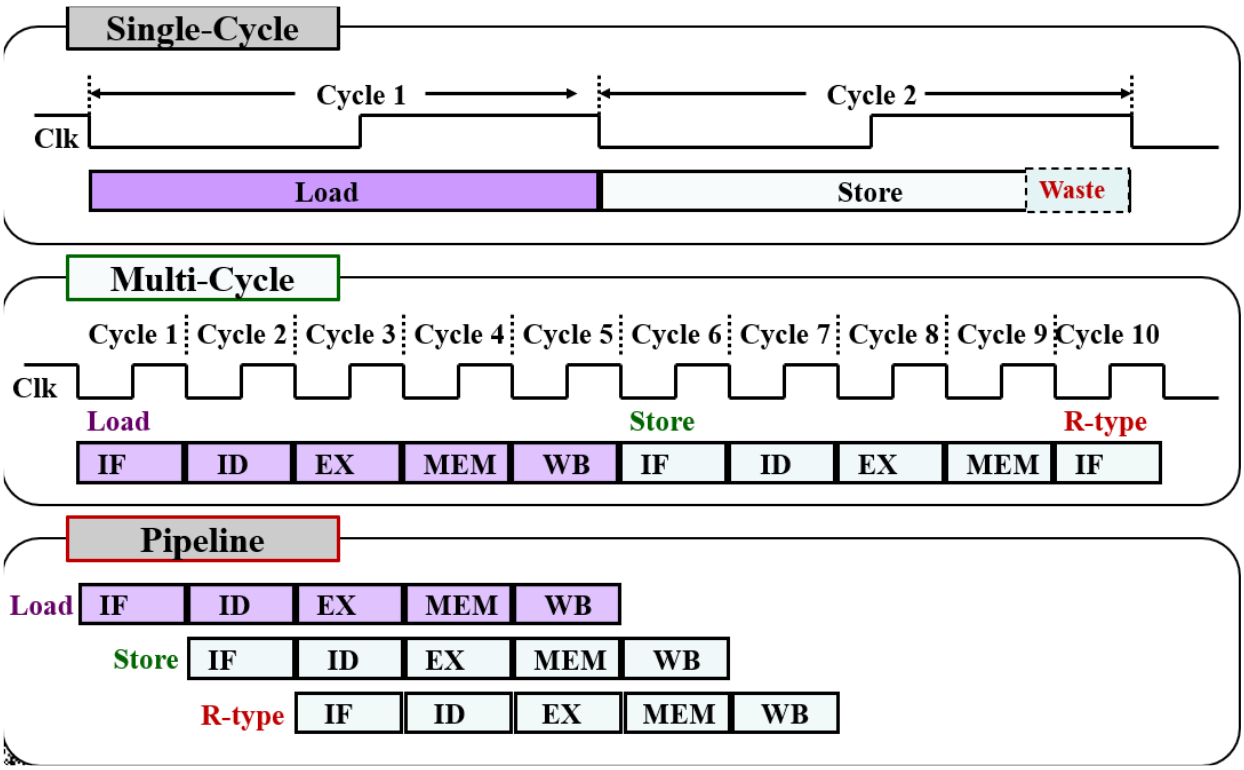


9.4.3 Datapath and Control



9.5 Pipeline Performance

Different Implementations:



9.5.1 Single Cycle Processor

Performance

Cycle time:

- $CT_{seq} = \sum_{k=1}^{N} T_k$
- $T_k =$ Time for operation in stage k
- $N =$ Number of stages

Total Execution Time for I instructions:

- $T_{seq} =$ Cycles $\times$ Cycle Time $= I \times CT_{seq} = I \times \sum_{k=1}^{N} T_k$

Example

Instruction	Inst Mem	Reg read	ALU	Data Mem	Reg write	Total
ALU	2	1	2		1	**6**
lw	2	1	2	2	1	**8**
sw	2	1	2	2		**7**
beq	2	1	2			**5**

Cycle time:

- Choose the longest total time = $8\ ns$
- To execute 100 instructions:

  $100 \times 8\ ns = 800\ ns$

9.5.2 Multi-Cycle Processor

Performance

Cycle time:

- $CT_{multi} = max(T_k)$
- $max(T_k) =$ longest stage duration among the N stages

Total Execution Time for I instructions:

- $T_{multi} =$ Cycles $\times$ Cycle Time $= I \times$ Average CPI $\times CT_{multi}$
- Average CPI is needed because each instruction takes different number of cycles to finish

Example

Instruction	Inst Mem	Reg read	ALU	Data Mem	Reg write	Total
ALU	2	1	2		1	6
lw	2	1	2	2	1	8
sw	2	1	2	2		7
beq	2	1	2			5

Cycle Time:

- Choose the longest stage time = $2\ ns$

To execute 100 instructions, with a given average CPI of 4.6:

- $100 \times 4.6 \times 2\ ns = 920\ ns$

9.5.3 Pipeline Processor

Performance

Cycle Time:

- $CT_{pipeline} = max(T_k) + T_d$
- $max(T_k) =$ longest stage duration among the N stages
- $T_d =$ Overhead for pipelining, e.g. pipeline register

Cycles needed for I instructions:

- $I + N - 1$
- $N - 1$ is the cycles wasted in filling up the pipeline

Total Time needed for I instructions:

- $T_{pipeline} = \text{Cycle} \times CT_{pipeline} = (I + N - 1) \times (max(T_k) + T_d)$

Example

Instruction	Inst Mem	Reg read	ALU	Data Mem	Reg write	Total
ALU	2	1	2		1	**6**
lw	2	1	2	2	1	**8**
sw	2	1	2	2		**7**
beq	2	1	2			**5**

Cycle Time:

- Assume pipeline register delay of $0.5\ ns$
- Longest stage time + overhead = $2 + 0.5 = 2.5\ ns$

To execute 100 instructions:

- $(100 + 5 - 1) \times 2.5\ ns = 260\ ns$

9.5.4 Ideal Speedup

Assumptions for ideal case:

- Every stage takes the same amount of time: $\sum_{k=1}^{N} T_k = N \times T_k$
- No pipeline overhead -> $T_d = 0$
- Number of instructions I, is much larger than number of stages N

Note: The above also shows how pipeline processor loses performance

$$
\begin{aligned}
Speedup_{pipeline} &= \frac{T_{seq}}{T_{pipeline}} \\
&= \frac{I \times \sum_{k=1}^{N} T_k}{(I + N - 1) \times (max(T_k) + T_d)} \\
&= \frac{I \times N \times T_k}{(I + N - 1) \times T_k} \\
&\approx \frac{I \times N \times T_k}{I \times T_k} \\
&\approx N
\end{aligned}
$$

> Conclusion: Pipeline processor can gain N times speedup, where N is the number of pipeline stages

> 流水线处理器相对于顺序（非流水线）处理器的性能加速。这里使用了数学公式来阐明这种加速是如何计算的，并指出在什么条件下可以达到理想的加速。我们将逐步解释这个过程。

假设条件：

1. **每个阶段的时间相同：** 这意味着流水线的每个阶段都需要相同的时间来完成其任务，表示为 T_k。
2. **没有流水线开销：** 也就是说，不存在因为指令在流水线中移动而产生的额外时间延迟或者是需要额外的周期。这被表示为 $T_d = 0$（这里的 T_d 是指流水线开销）。
3. **指令数量远大于阶段数：** 这意味着执行的指令数量 I 远大于流水线的阶段数 N，从而任何与流水线启动和结束相关的开销相对于整个执行过程是可以忽略不计的。

加速计算：

首先，我们定义了顺序执行时间 T_{seq} 和流水线执行时间 $T_{pipeline}$，然后我们计算两者的比值以得到加速比 $Speedup_{pipeline}$。

1. **顺序执行时间** 是所有指令在非流水线设置中执行所需的总时间，计算为指令数 I 乘以每条指令的执行时间（由于每个阶段都花费 T_k 时间，总时间为 $N \times T_k$）。
2. **流水线执行时间** 是在流水线设置中执行所有指令所需的时间。重要的是要注意，第一条指令完成之后，每个额外的周期都会完成一条新指令。因此，总时间包括了初始的 N 个阶段和随后的 $I - 1$ 个周期（每条指令一个），再加上流水线的开销时间 T_d，但在我们的理想假设中，$T_d = 0$。
3. **加速比** 是顺序时间与流水线时间的比率。在这种情况下，因为每个阶段花费相同的时间并且没有额外的流水线开销，这个比率简化为 $I \times N \times T_k$ 除以 $I \times T_k$（因为 N 阶段花费 $N \times T_k$ 时间，总共有 I 条指令）。当指令数量 I 很大时，$I + N - 1$ 约等于 I，因此加速比接近 N。

结论：

在理想情况下，流水线处理器可以实现 N 倍的加速，其中 N 是流水线的阶段数。这意味着如果您有一个5阶段的流水线，理论上您的处理器速度可以提高5倍。然而，这个理想情况很少出现，因为实际的流水线执行通常会遇到各种开销和延迟，例如由于数据依赖性、控制依赖性、资源冲突等引起的流水线停顿。所以，实际的加速比通常会低于理想情况。