

Pipelined MIPS

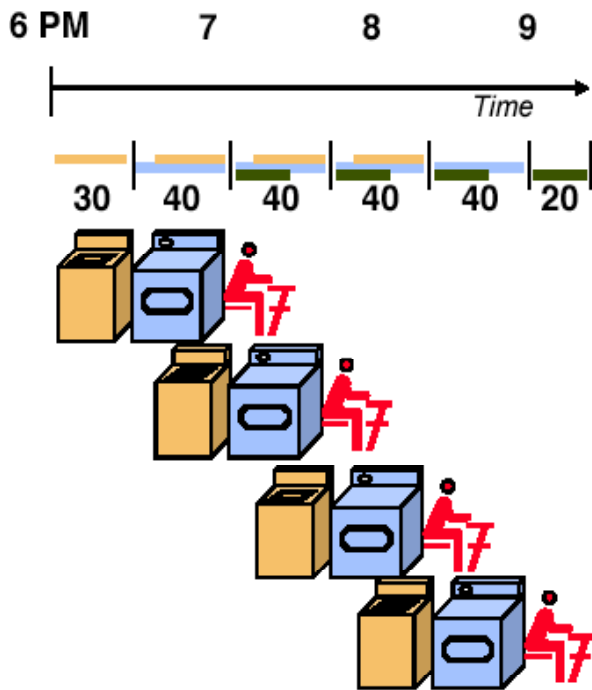
Why pipelining?

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

How is it possible? By overlapping the execution of consecutive instructions ...

Study the Laundromat example from the book.

Example of pipelining



- ❑ Pipelining doesn't help **latency** of single load, it helps **throughput** of entire workload
- ❑ Pipeline rate limited by **slowest** pipeline stage
- ❑ **Multiple** tasks operating simultaneously using different resources
- ❑ Potential speedup = **Number of pipe stages**
- ❑ Unbalanced lengths of pipe stages reduces speedup
- ❑ Time to **"fill"** pipeline and time to **"drain"** it reduces speedup

Pipelining in a laundry

Washer takes 30 minutes

Dryer takes 40 minutes

Folding takes 20 minutes

Instruction execution review

- ❑ Executing a MIPS instruction can take up to five steps.

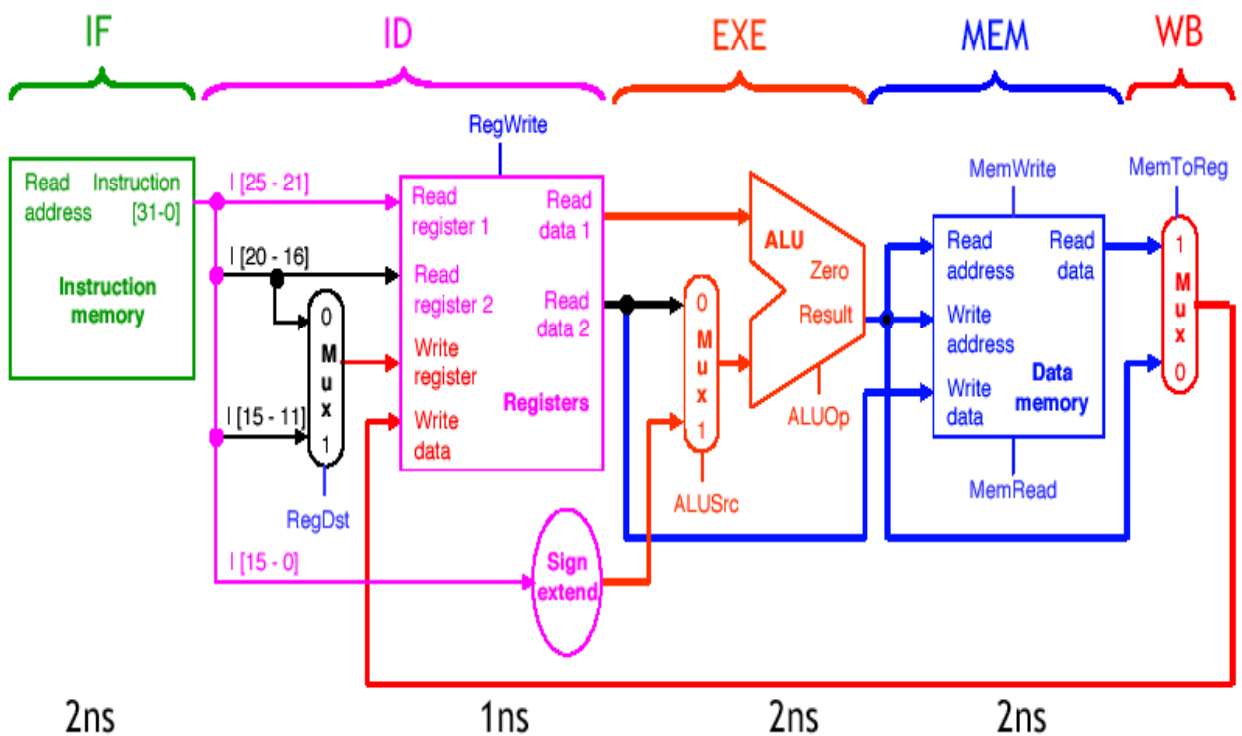
Step	Name	Description
Instruction Fetch	IF	Read an instruction from memory.
Instruction Decode	ID	Read source registers and generate control signals.
Execute	EX	Compute an R-type result or a branch outcome.
Memory	MEM	Read or write the data memory.
Writeback	WB	Store a result in the destination register.

- ❑ However, as we saw, not all instructions need all five steps.

Instruction	Steps required					
beq	IF	ID	EX			
R-type	IF	ID	EX			WB
sw	IF	ID	EX	MEM		
lw	IF	ID	EX	MEM		WB

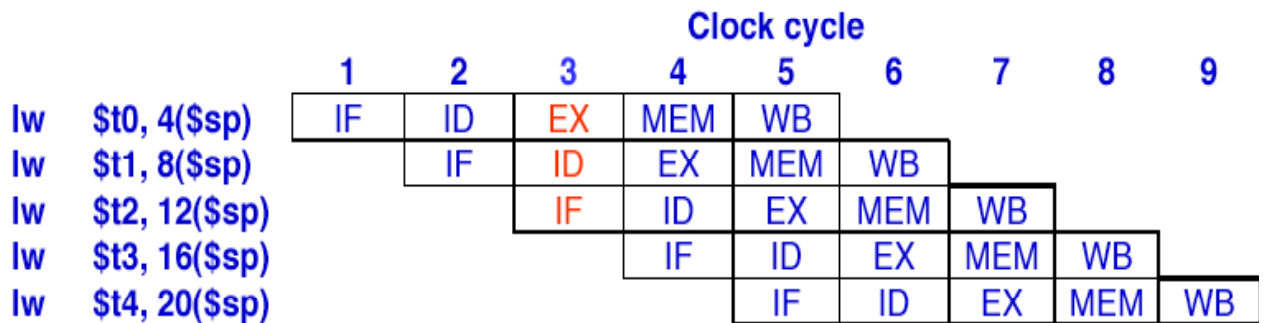
Break datapath into 5 stages

- ❑ 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.

Pipelining Loads



- ❑ **A pipeline diagram shows the execution of a series of instructions.**
 - The instruction sequence is shown vertically, from top to bottom.
 - Clock cycles are shown horizontally, from left to right.
 - Each instruction is divided into its component stages. (We show five stages for every instruction, which will make the control unit easier.)

- ❑ **This clearly indicates the overlapping of instructions. For example, there are three instructions active in the third cycle above.**
 - The “lw \$t0” instruction is in its Execute stage.
 - Simultaneously, the “lw \$t1” is in its Instruction Decode stage.
 - Also, the “lw \$t2” instruction is just being fetched.

Since multiple memory operations overlap, we had to return to Harvard architecture!

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.

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

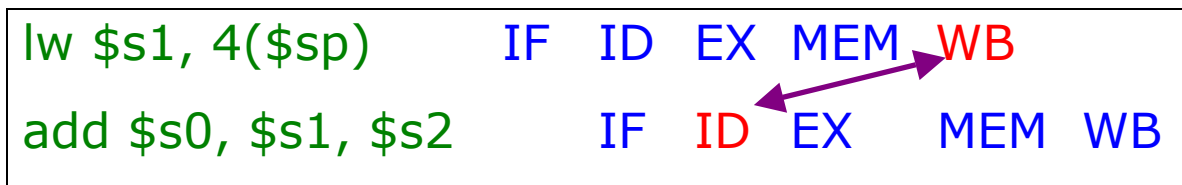
If there are k stages, and each stage takes t time units, then the time needed to execute N instructions is

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

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:



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 to 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	nop	ID

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

Hazards slow down the instruction execution speed.

Control hazard

sub \$s1, \$t1, \$t2	IF	ID	EX	MEM	WB
beq \$s1, \$zero L		IF	ID	EX	MEM
some instruction here			IF	ID	EX

Will the correct instruction be fetched?

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)

IR := Memory[PC]; PC := PC + 4

(Instruction decode and Register fetch)

A := Reg[IR[25:21]], B := Reg[IR[20:16]]

ALUout := PC + sign-extend(IR[15:0])

(Execute | Memory address | Branch completion)

Memory refer: ALUout := A + IR[15:0]

R-type (ALU): ALUout := A op B

Branch: if A = B then PC := ALUout

(Memory access | R-type completion)

LW: MDR := Memory[ALUout]

SW: Memory[ALUout] := B

R-type: Reg[IR[15:11]] := ALUout

(Write back)

LW: Reg[[20:16]] := MDR

sub \$s1, \$t1, \$t2	IF	ID	EX	MEM	WB	
beq \$s1, \$zero L		IF	ID	EX	MEM	
Some instruction here			IF	0	IF	ID

No action performed here

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