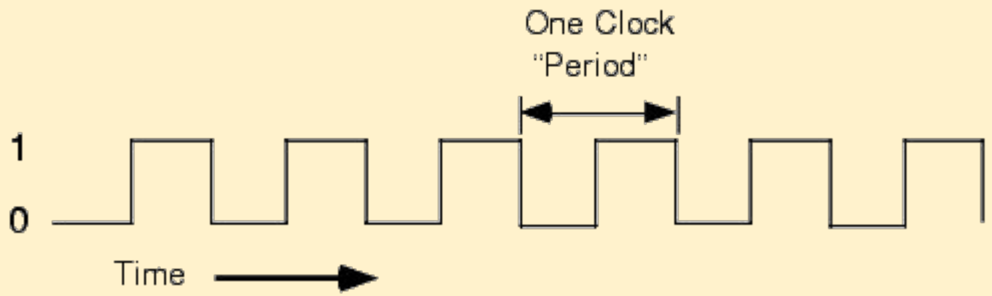
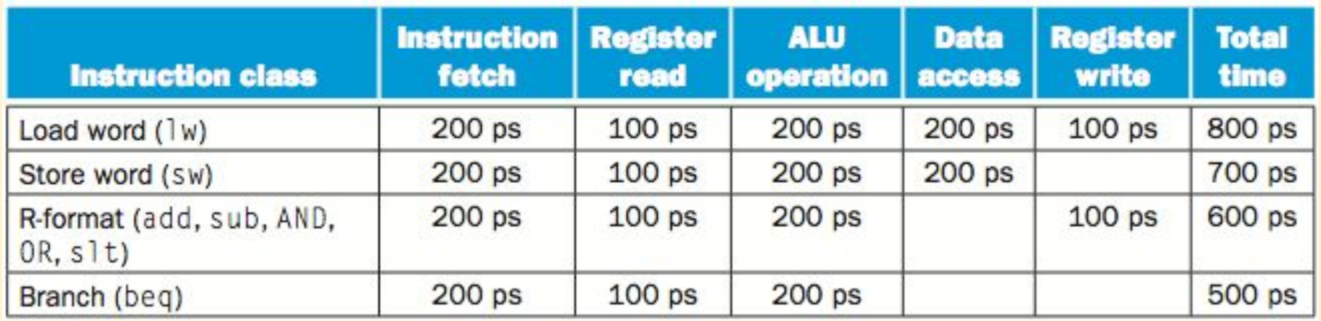
Drawbacks of simple implementation

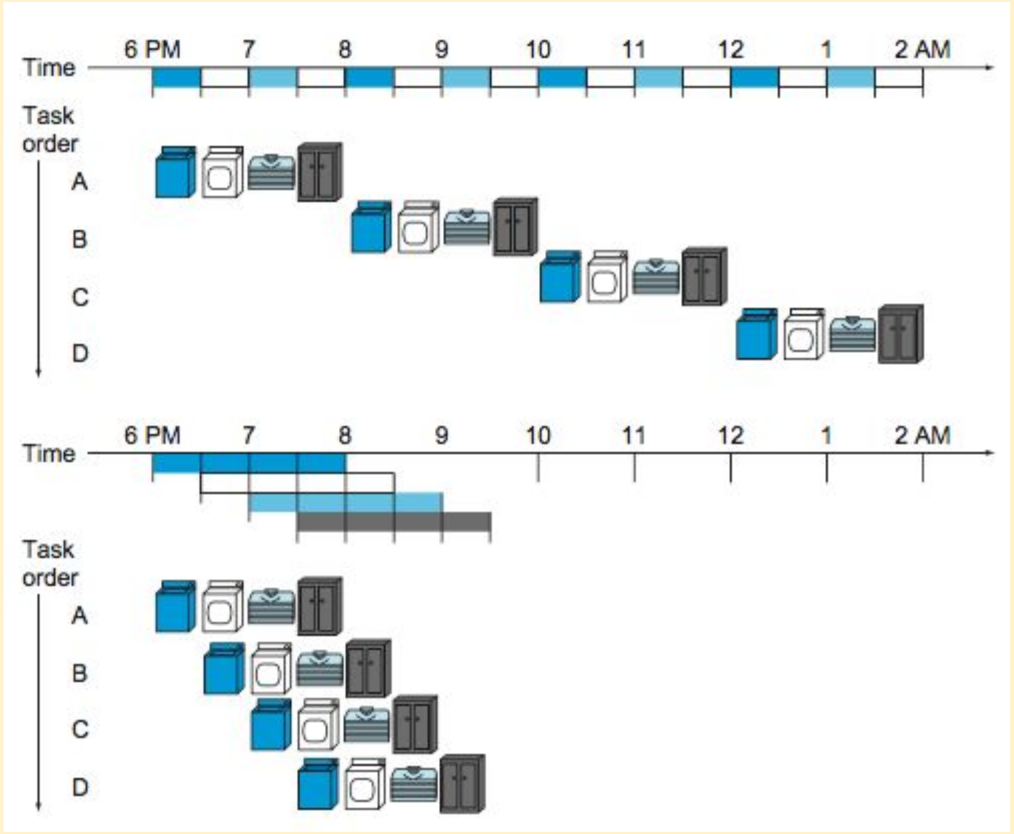




Instruction class	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
Load word (lw)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store word (sw)	200 ps	100 ps	200 ps	200 ps		700 ps
R-format (add, sub, AND, OR, slt)	200 ps	100 ps	200 ps		100 ps	600 ps
Branch (beq)	200 ps	100 ps	200 ps			500 ps

Pipelining analogy

- takes 8 hours for 4 loads in non-pipelined approach
- takes 3.5 hours for 4 loads in pipelined approach
- 4 loads will have a speedup of 8/3.5 = 2.3
- for infinite laundry loads, speedup approaches 4, the number of stages



MIPS pipeline

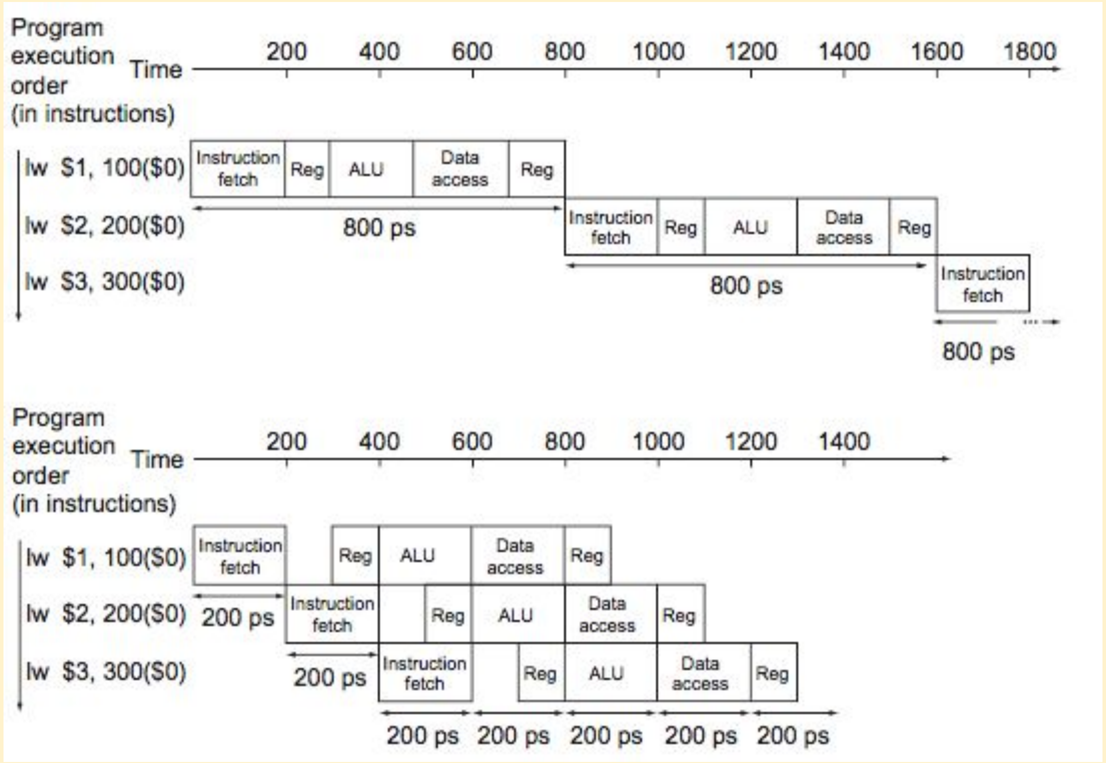
Five states, one (much faster) clock tick per stage

1. IF: instruction fetch from memory
2. ID: instruction decode and register read
3. EX: ALU executes operation or calculates address
4. MEM: access data memory
5. WB: write results back to register file

Pipeline performance

- top, single-cycle, T=800 ps
- bottom, pipelined, T=200 ps

- top: each instruction executes in 800 ps
- bottom: each instruction executes in 1000 ps, but throughput is improved



Pipeline speedup

If all stages are balanced, speedup is:

$$\text{Time between instructions}_{\text{pipelined}} = \frac{\text{Time between instruction}_{\text{nonpipelined}}}{\text{Number of pipe stages}}$$

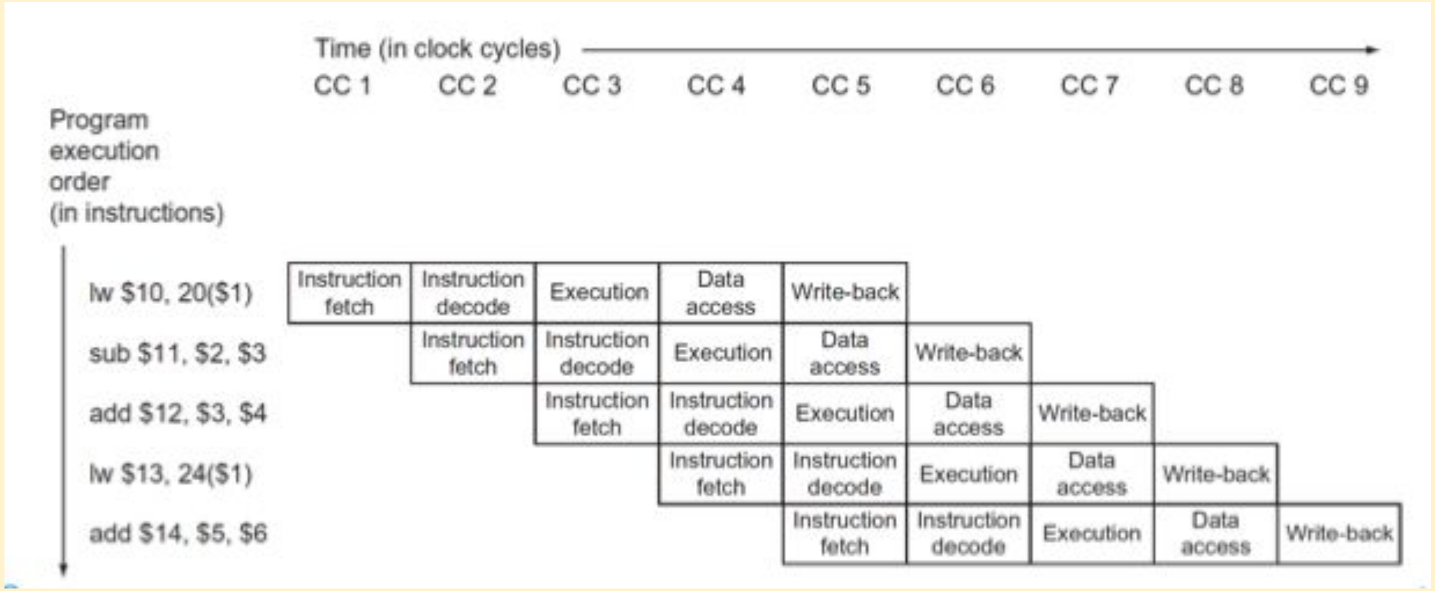
Ideally, a 5-stage pipeline will approach a 5 times speedup. The latency (time for each instruction), does not decrease.

Pipelining and ISA design

MIPS ISA is designed for pipelining

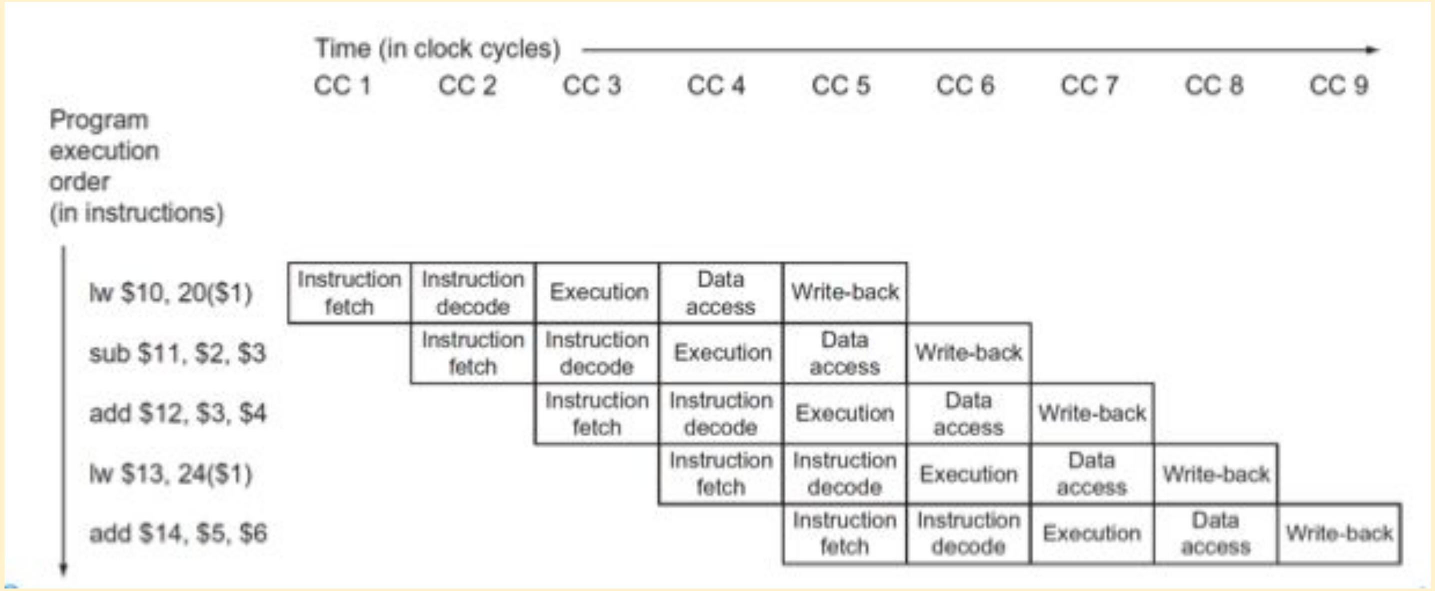
- all instructions are 32 bits which makes it easier to fetch and decode
- few instruction formats makes it faster to decode and read registers
- load/store addressing - ALU cannot have a memory operand
- memory alignment - words are aligned, makes access faster

MIPS pipelining



	CC 1	CC 2	CC 3	CC 4	CC 5	CC 6	CC 7	CC 8	CC 9
lw $10, 20($1)	Instruction fetch	Instruction decode	Execution	Data access	Write-back				
sub $11, $2, $3		Instruction fetch	Instruction decode	Execution	Data access	Write-back			
add $12, $3, $4			Instruction fetch	Instruction decode	Execution	Data access	Write-back		
lw $13, 24($1)				Instruction fetch	Instruction decode	Execution	Data access	Write-back	
add $14, $5, $6					Instruction fetch	Instruction decode	Execution	Data access	Write-back

What could go wrong?



Time (in clock cycles) →

Program execution order (in instructions) ↓

	CC 1	CC 2	CC 3	CC 4	CC 5	CC 6	CC 7	CC 8	CC 9
lw $10, 20($1)	Instruction fetch	Instruction decode	Execution	Data access	Write-back				
sub $11, $2, $3		Instruction fetch	Instruction decode	Execution	Data access	Write-back			
add $12, $3, $4			Instruction fetch	Instruction decode	Execution	Data access	Write-back		
lw $13, 24($1)				Instruction fetch	Instruction decode	Execution	Data access	Write-back	
add $14, $5, $6					Instruction fetch	Instruction decode	Execution	Data access	Write-back

Hazards

Situations that prevent starting the next instruction in the next cycle

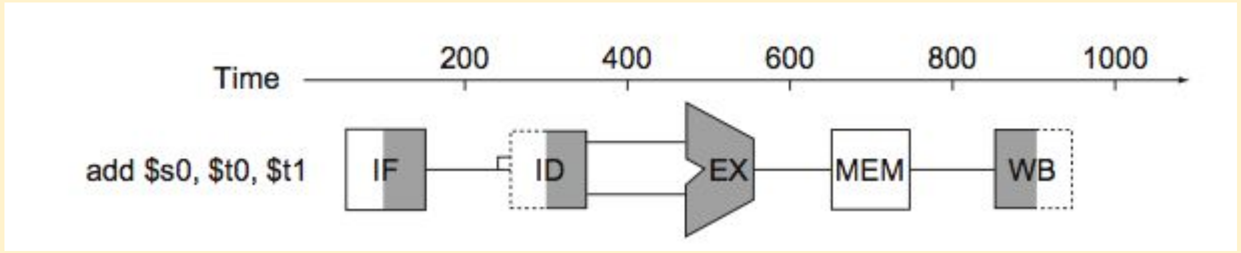
3 types:

1. structure hazard - a required resource is busy (doesn't happen in MIPS but would in a system that combined data/instruction memory)
2. data hazard - need to wait for previous instruction to complete its data access
3. control hazard - we don't know yet if we are branching or not

Pipeline phases

Shading on left means write, shading on right means read

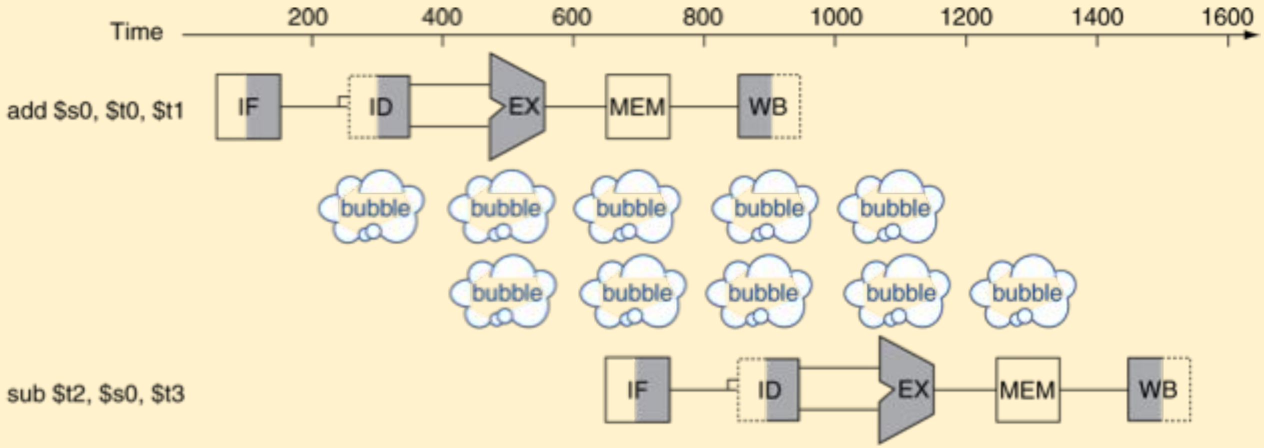
No shading means that component is not used. Full shading means it is busy the whole clock cycle.



Data hazards

An instruction has to wait for data from a previous instruction.

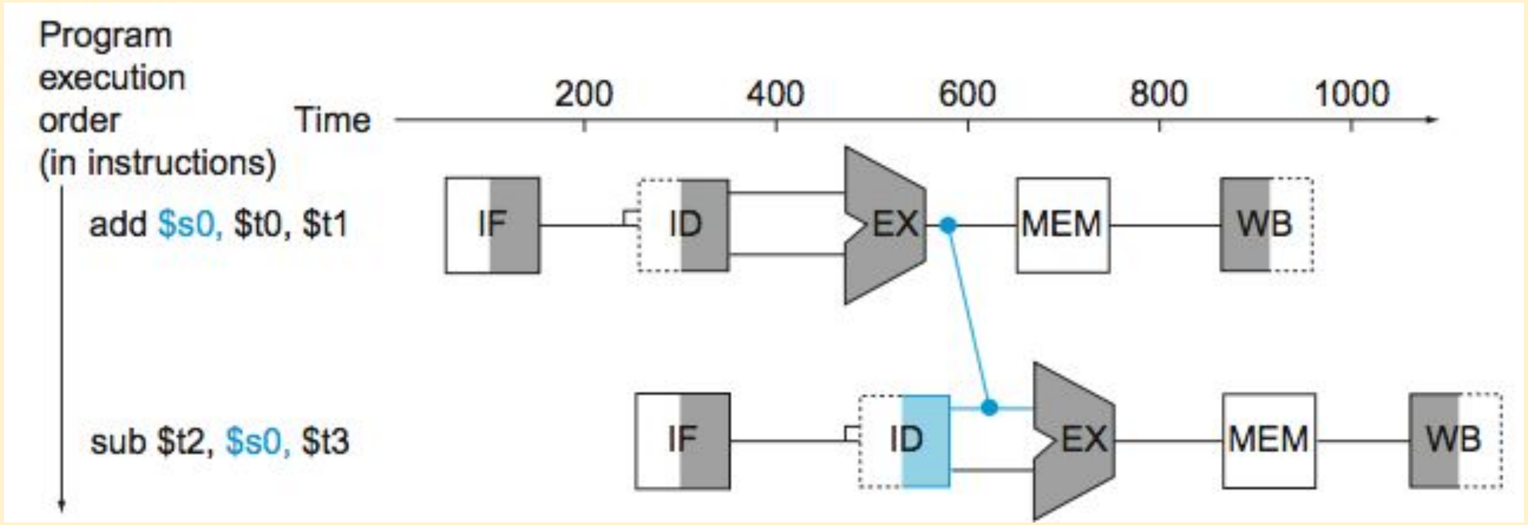
```
add   $s0, $t0, $t1
sub   $t2, $s0, $t3
```



Forwarding (bypassing)

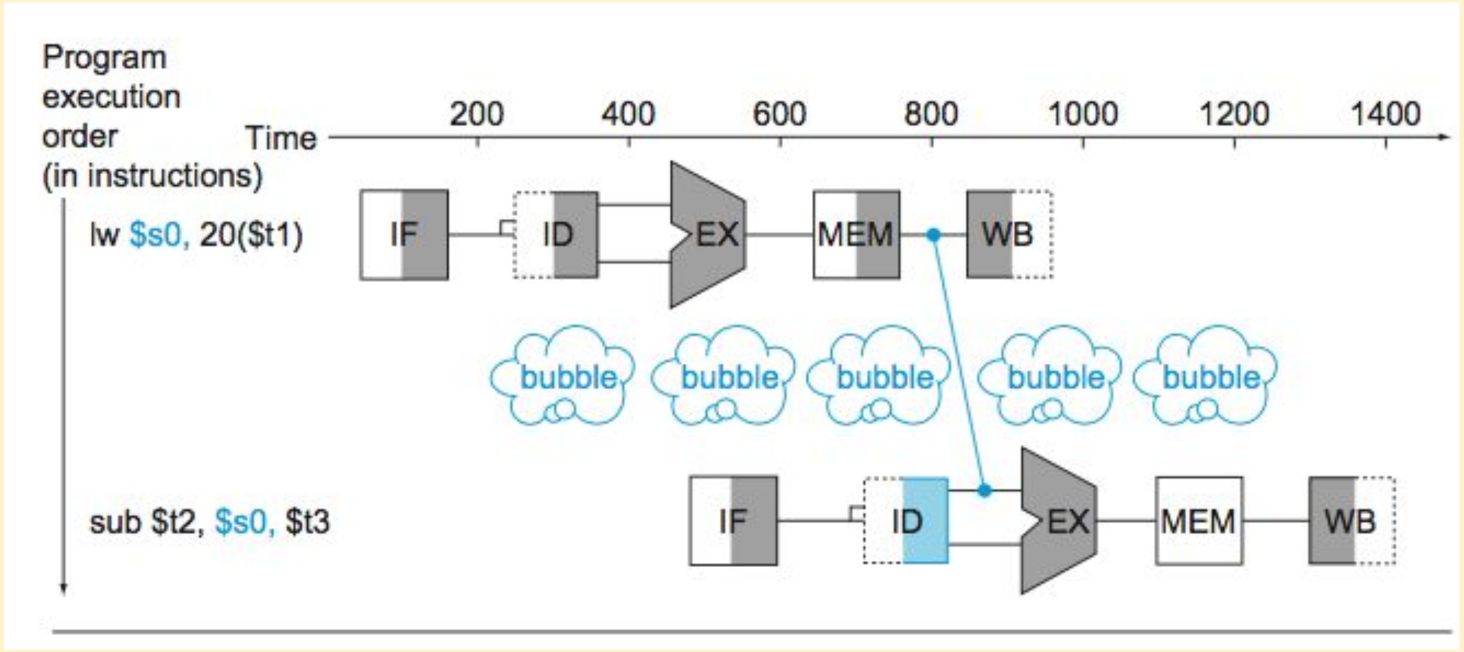
Use result as soon as it is computed, rather than wait for register WB

Requires extra connections in the data path



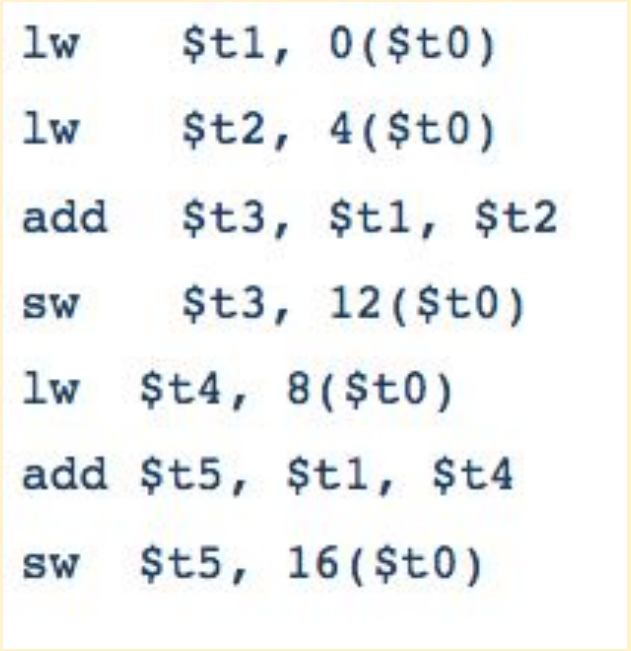
load-use hazard

Can't avoid load-use stall by forwarding, but forwarding limits stall to 1 cycle



Where are the hazards in this code?

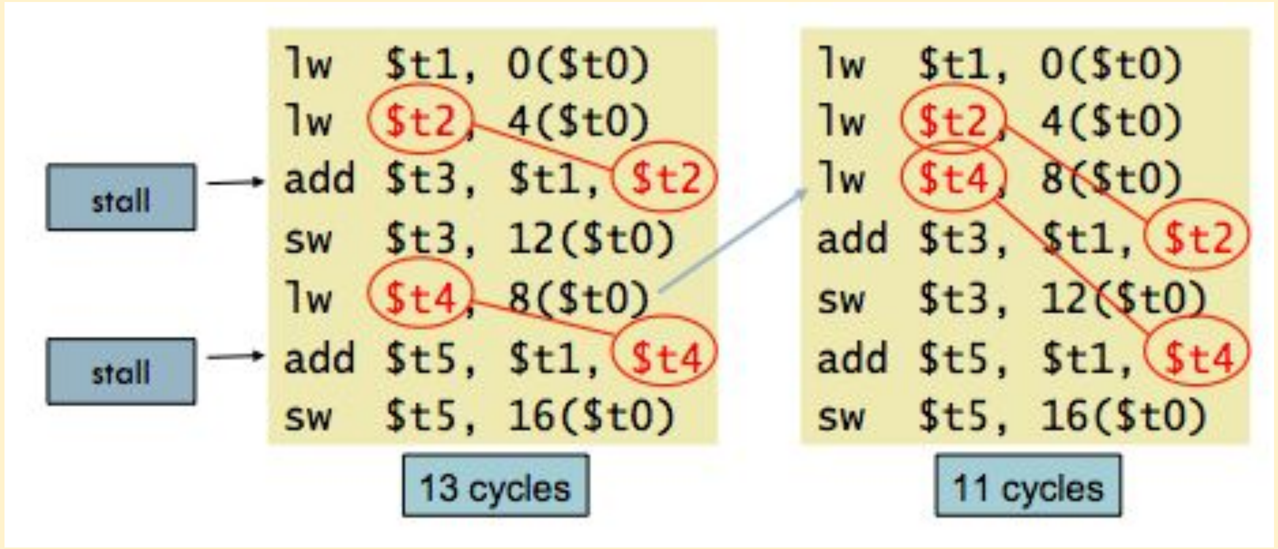
Can we reorder to code to avoid hazards?



```
lw    $t1, 0($t0)
lw    $t2, 4($t0)
add   $t3, $t1, $t2
sw    $t3, 12($t0)
lw  $t4, 8($t0)
add $t5, $t1, $t4
sw  $t5, 16($t0)
```

Code scheduling to avoid stalls

Reorder the code to avoid load-use hazards.



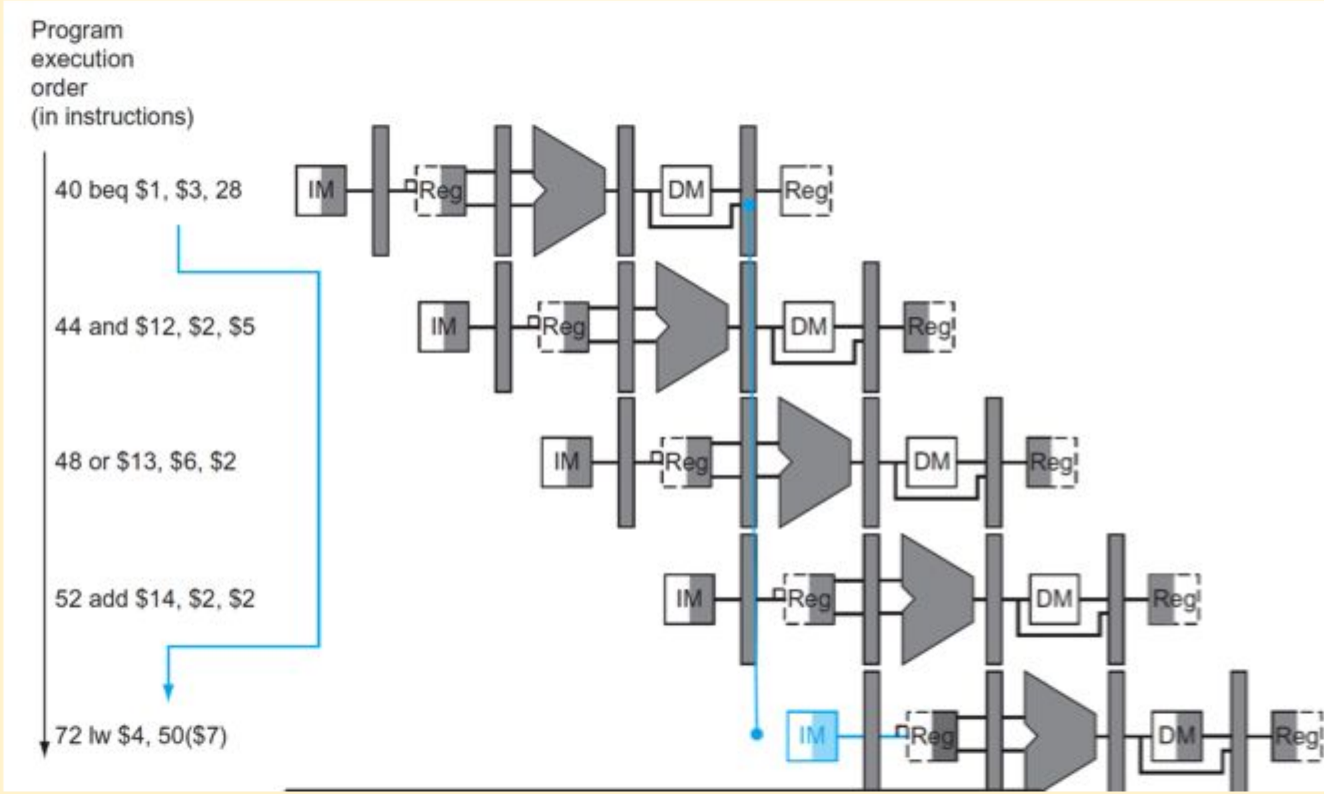
```
lw   $t1, 0($t0)
lw   $t2, 4($t0)
add  $t3, $t1, $t2
sw   $t3, 12($t0)
lw   $t4, 8($t0)
add  $t5, $t1, $t4
sw   $t5, 16($t0)
```

```
lw   $t1, 0($t0)
lw   $t2, 4($t0)
lw   $t4, 8($t0)
add  $t3, $t1, $t2
sw   $t3, 12($t0)
add  $t5, $t1, $t4
sw   $t5, 16($t0)
```

Control hazard

If the CPU assumes the branch is not taken but it is, it will have to bubble out the instructions that have started execution, and start executing at the correct instruction.

This would waste a lot of clock cycles.



Control hazards

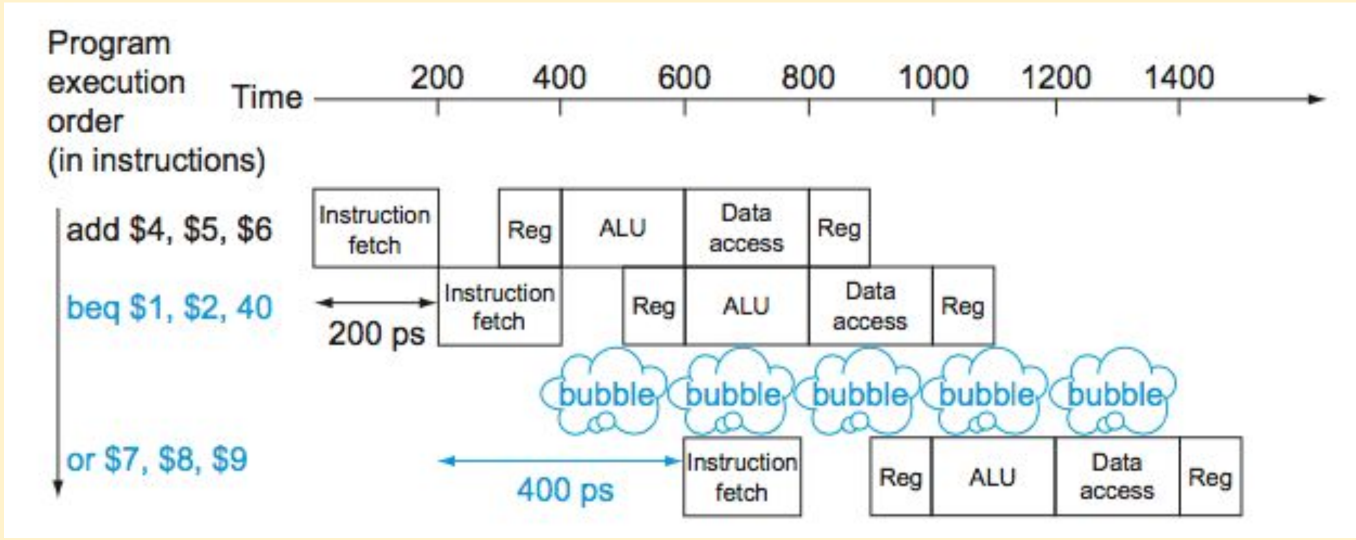
After a branch instruction, which instruction should be fetched, the one immediately following the branch or the one at the target?

In order to minimize the number of clock cycles wasted, add some circuitry in the ID stage to determine if we are branching or not.

Stall on branch

Branch instructions often cause a stall because the CPU does not know whether to branch or not.

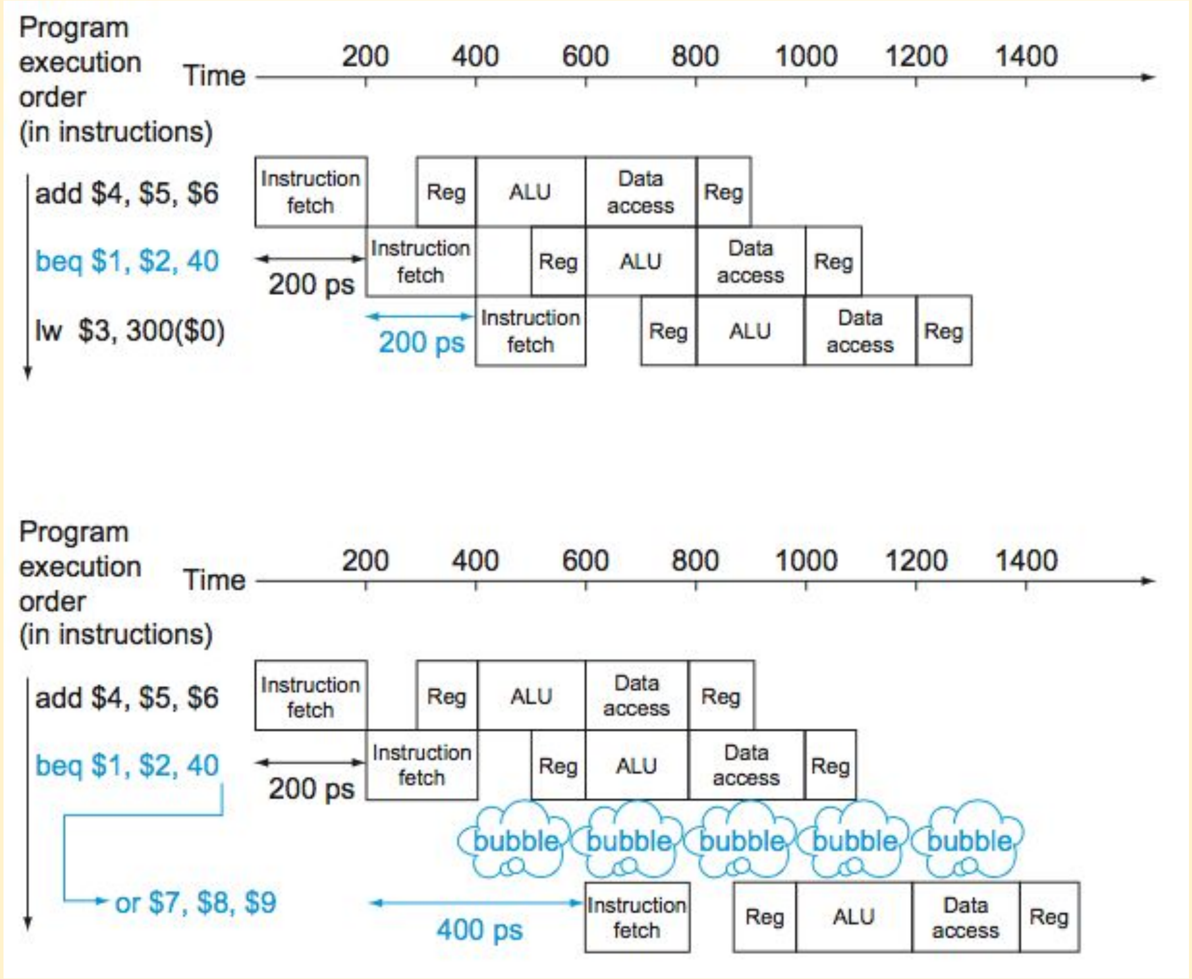
SPEC benchmarks: 17% of instructions were branches, so this would increase CPI by 17%.



Branch prediction

The CPU will predict that the branch is not taken. If that is correct (top diagram) then no stall is required.

If that prediction is wrong (bottom diagram) then only one stall is required. The instruction that had started to execute is bubbled, turned into a NOP.



Types of branch prediction

Static branch prediction (assembler)

- based on typical branch behavior at loops and conditional branches (ifs)
- predict backward branches are taken
- predict forward branches are not taken

Dynamic branch prediction (CPU)

- extra circuitry measures behavior for each branch
- then assumes branch or not based on past behavior
- if it is wrong, will stall and refetch, updating history
- can be up to 90% accurate

What MIPS does: delayed branching

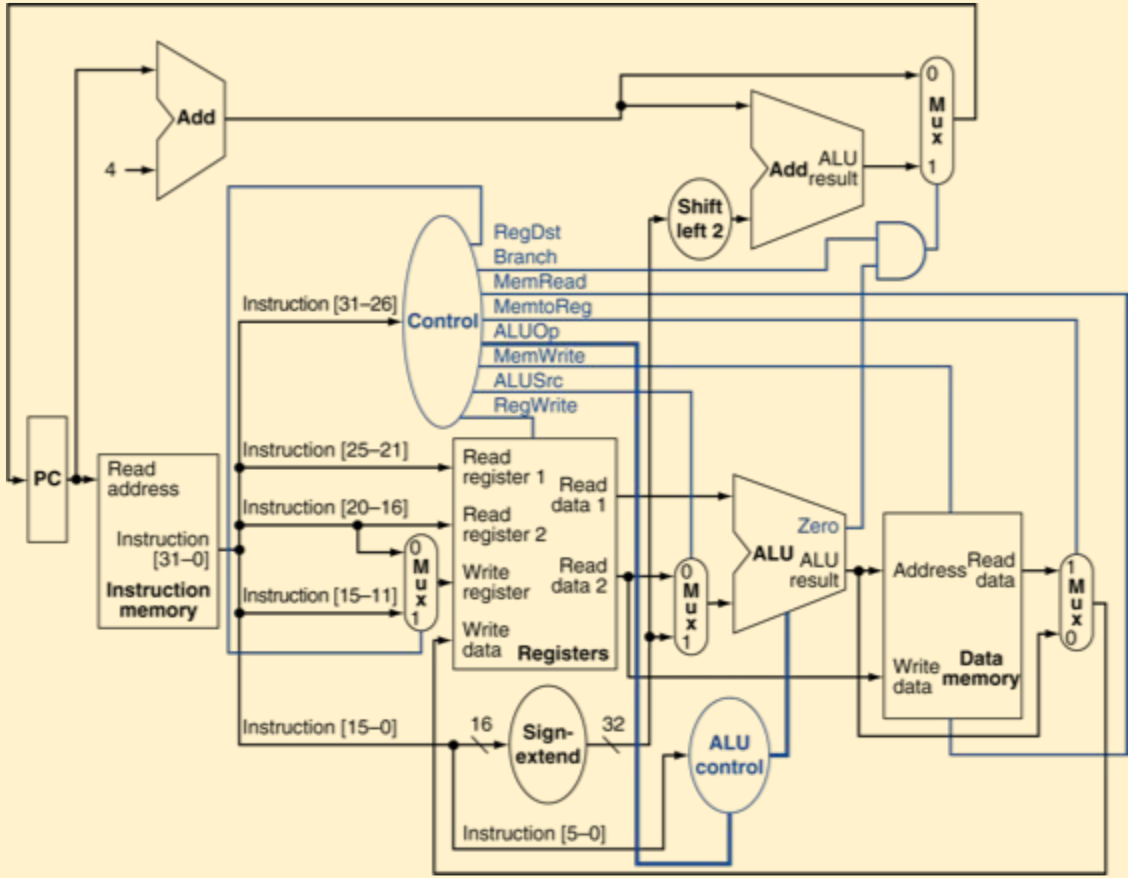
- implemented by MIPS assemblers
- the instruction following the branch is always executed
- if the branch should have been taken, it will be taken after that one extra instruction
- the assembler selects an instruction that is not affected by the branch to place after the branch

Pipelining

What we know so far:

- pipelining increases performance by increasing instruction throughput
  - works on multiple instructions at a time
  - each instruction has the same latency
- but pipelining is subject to hazards: structure, data, control
- ISA affects complexity of pipeline implementation

How to divide the data path?



Pipelined data path

In the single-cycle implementation, each instruction executed in one clock cycle

In the multi-cycle pipelined design, the clock will be faster and instructions execute in phases over several clock cycles

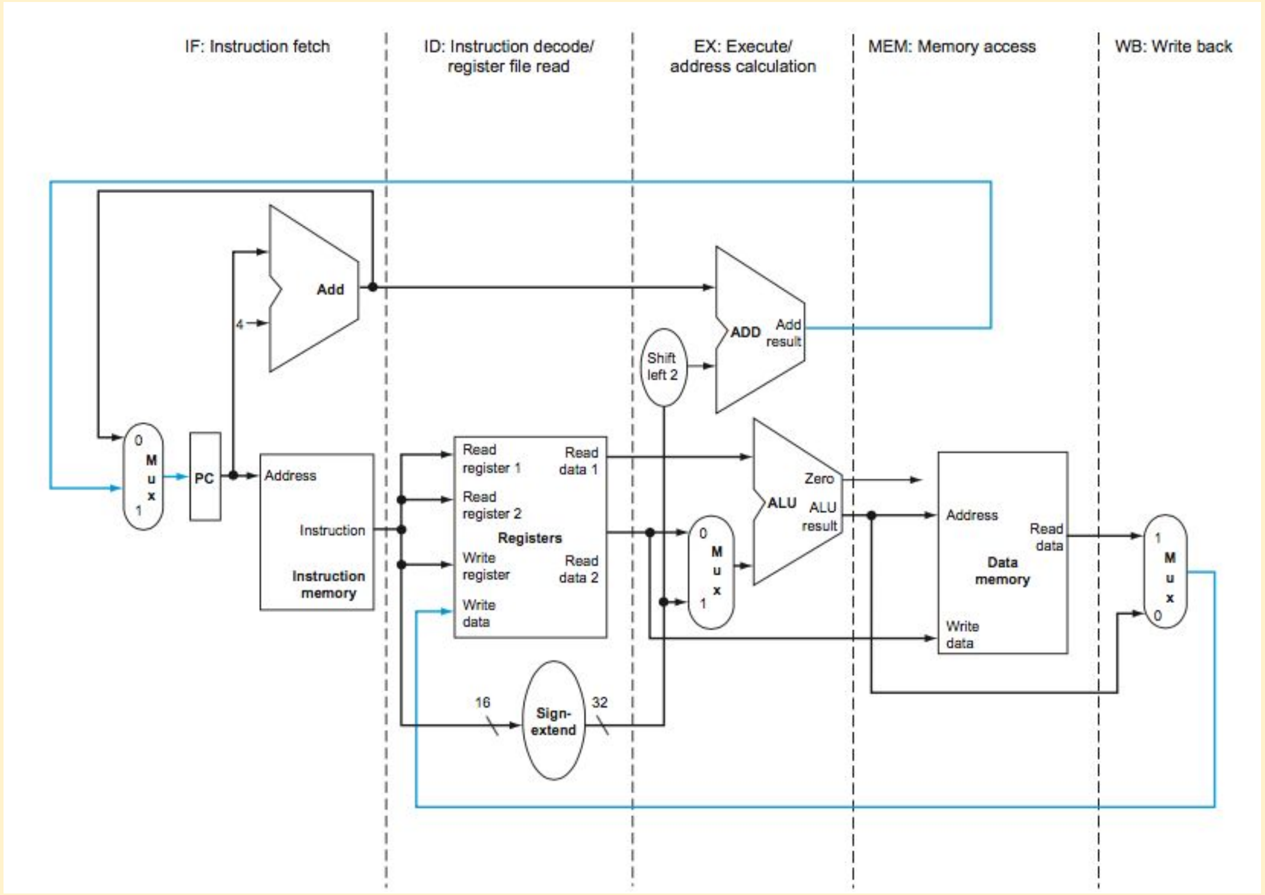
Each phase takes one clock cycle

We will redesign the CPU slightly so that each CPU section can operate independently on one instruction at the same time

Divide the data path into 5 sections

Data flows left to right with 2 exceptions shown in blue lines:

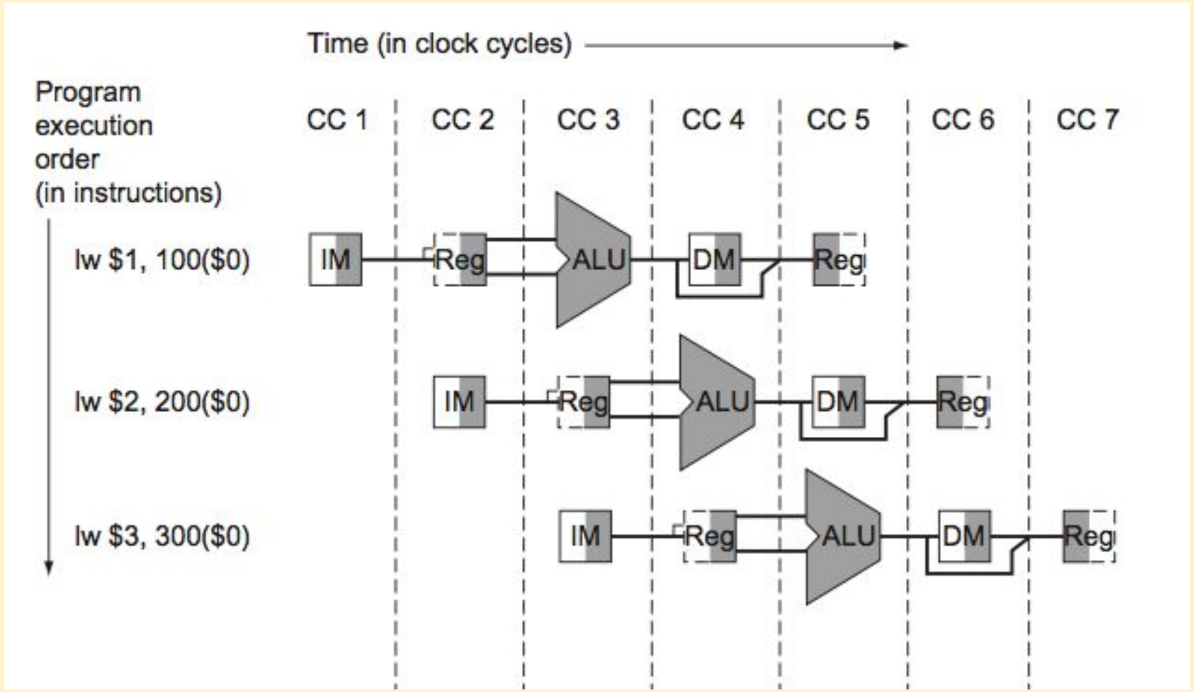
- register write back
- PC update



How instructions are processed concurrently

Shading: right side for read, left side for write

Notice that we can write to the register file and read from it in the same cycle



Pipeline architecture

A pipelined computer executes instructions concurrently

Hardware units are divided into stages:

- each stage takes 1 clock period
- stages are separated by pipeline registers that preserve and pass partial results to the next stage

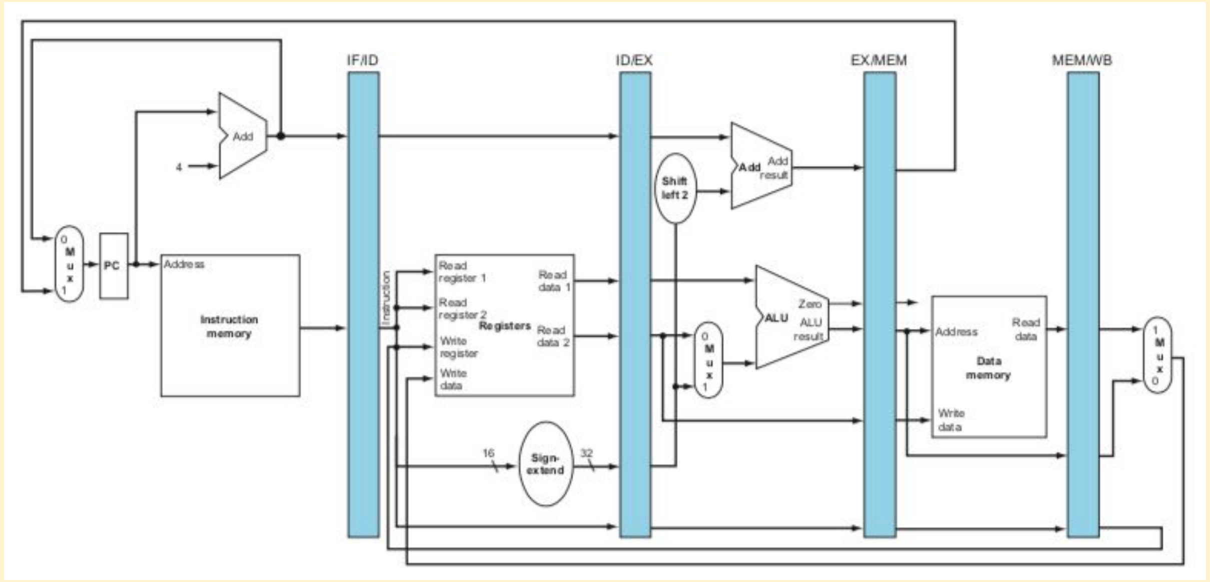
Pipeline registers

Pipeline registers are needed between stages to hold information produced in previous cycles.

5 stages, 4 pipeline registers

They are named according to the registers they are between.

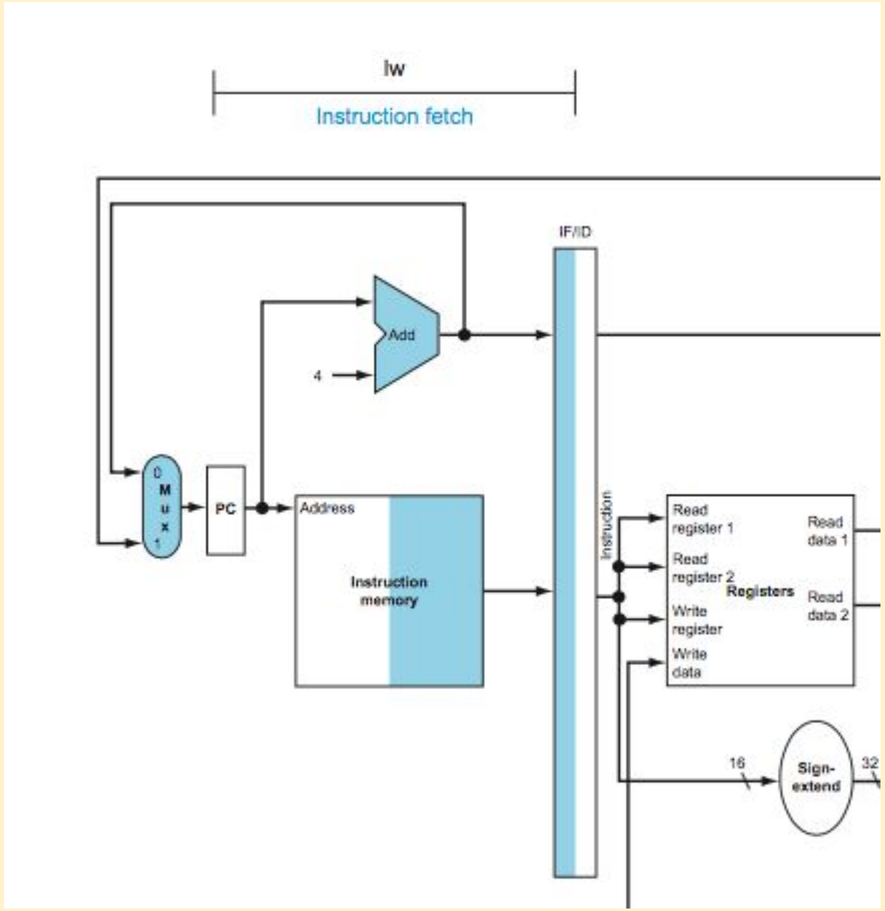
Next, we look at how the lw instruction would travel through these stages.



IF stage for lw

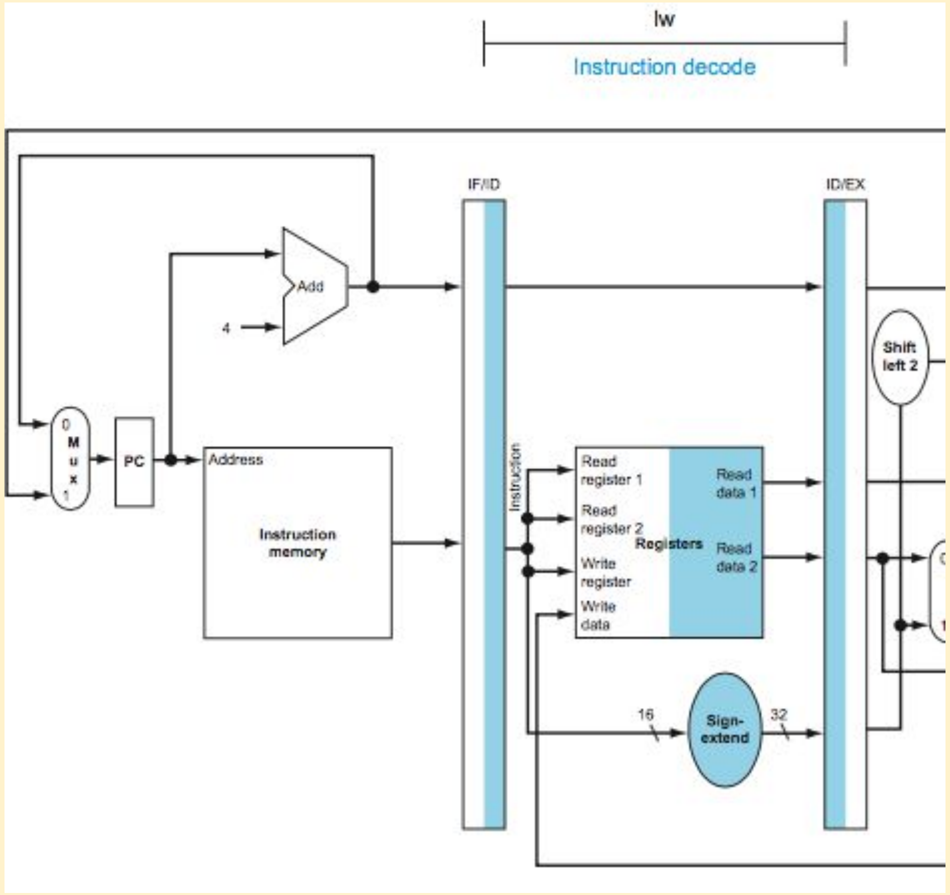
- instruction is read and placed into IF/ID register
- meanwhile the PC is updated by 4
- the PC is also saved in the pipeline register in case it is needed for a beq instruction

lines going through pipeline registers indicate information that is passed forward



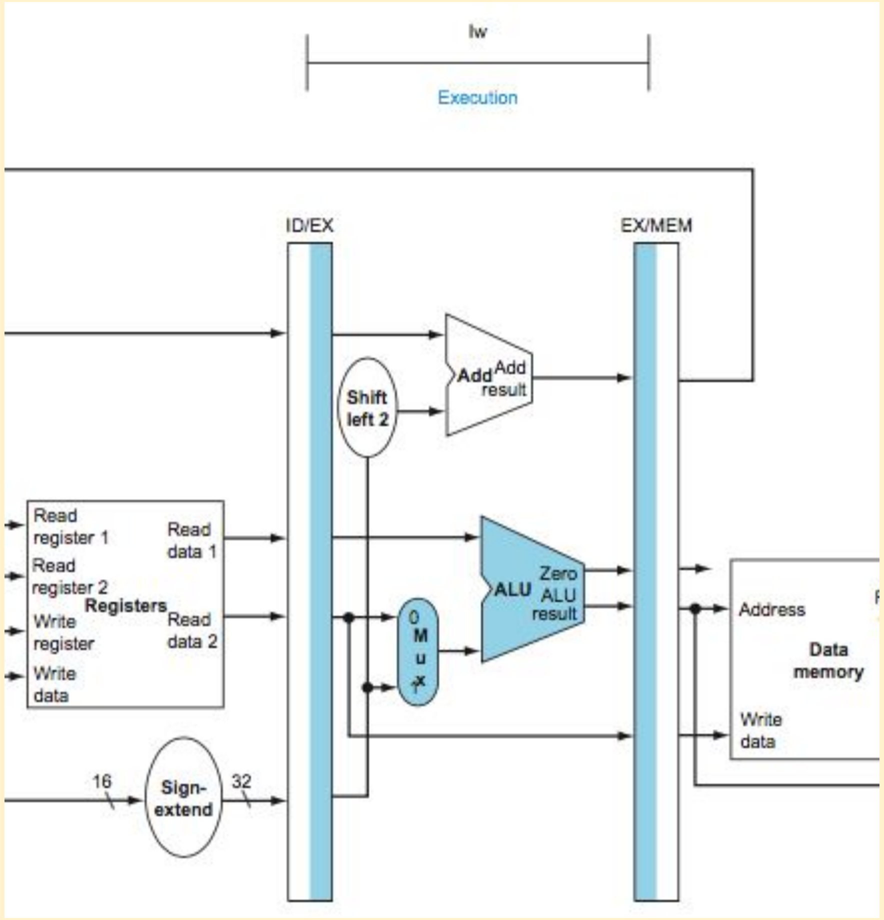
ID stage for lw

- the two registers, rt (destination) and rs (source) are read
- even though we don't need to read rt, the CPU doesn't know that yet
- the 16-bit immediate field is sign-extended to 32 bits
- the register read (rs) and sign-extended immediate field is stored in the ID/EX pipeline register



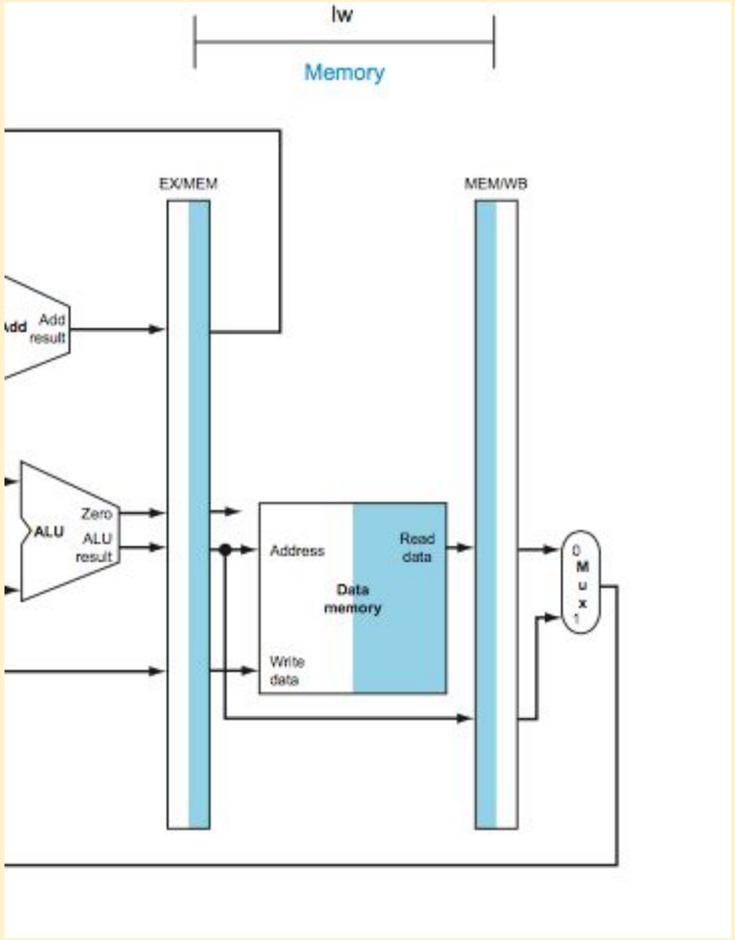
EXE stage for lw

- sum of rs and the sign-extended offset are summed
- the sum is placed on the EX/MEM register



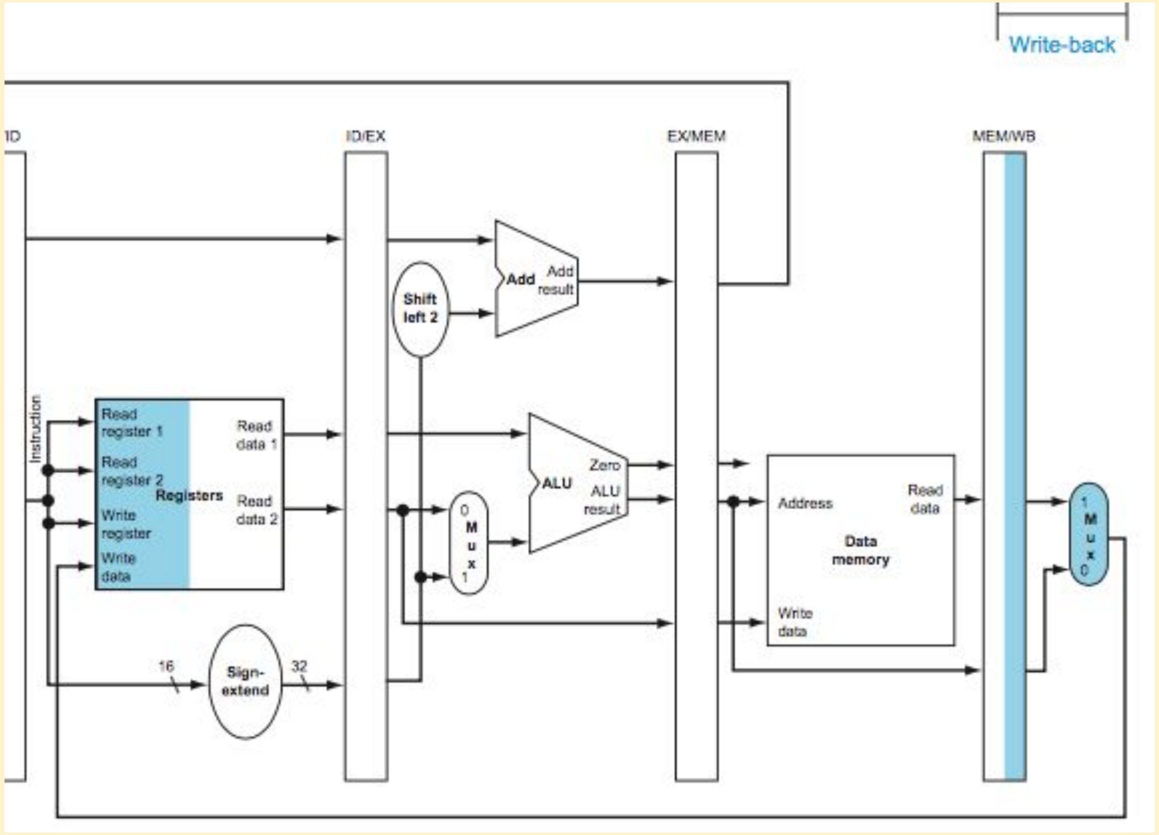
MEM stage for lw

- the address is picked up from the EX/MEM register
- that address is read
- the contents are placed on the MEM/WB register



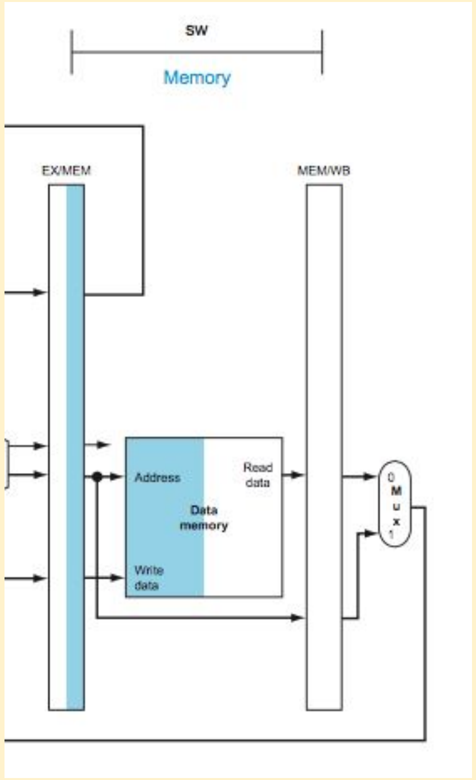
WB stage for lw

- data is read from the MEM/WB register and written back to rt in the register file



What about sw?

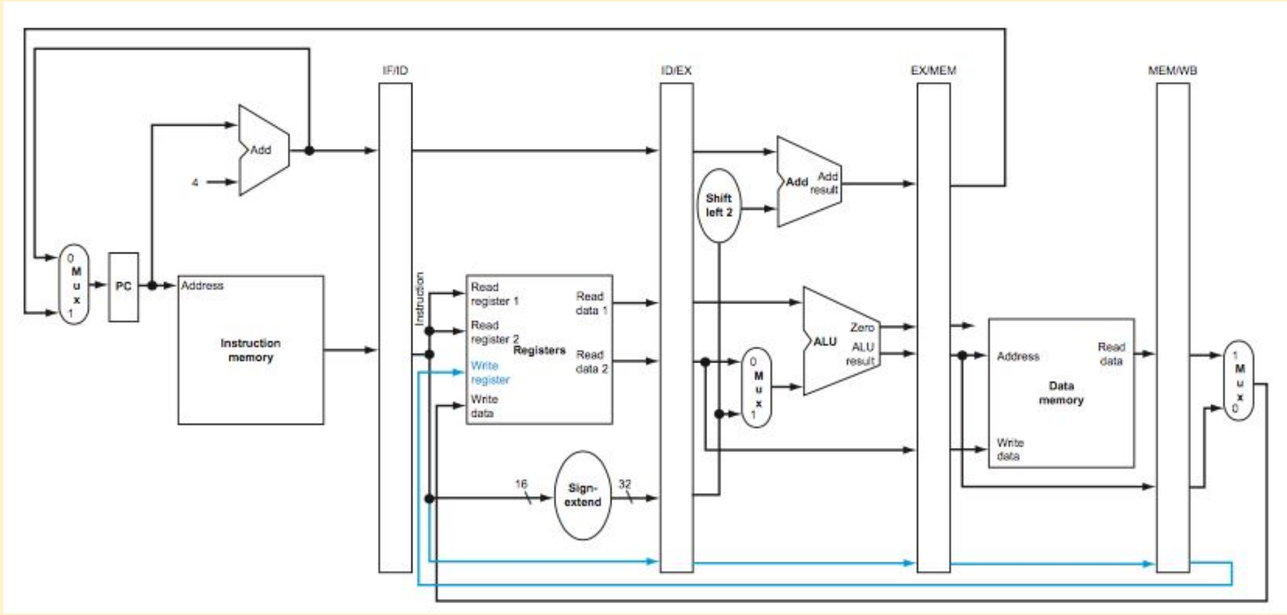
- the first 2 stages are the same
- in the EXE stage, the value to be written is carried forward on the pipeline registers
- in the MEM stage, we are writing to memory instead of reading it
- the data being written comes from the pipeline register, carried forward from the previous pipeline register
- nothing happens in the WB stage



Modification to lw

Which register to write to will get overwritten in the pipeline register as new instructions come into the pipeline.

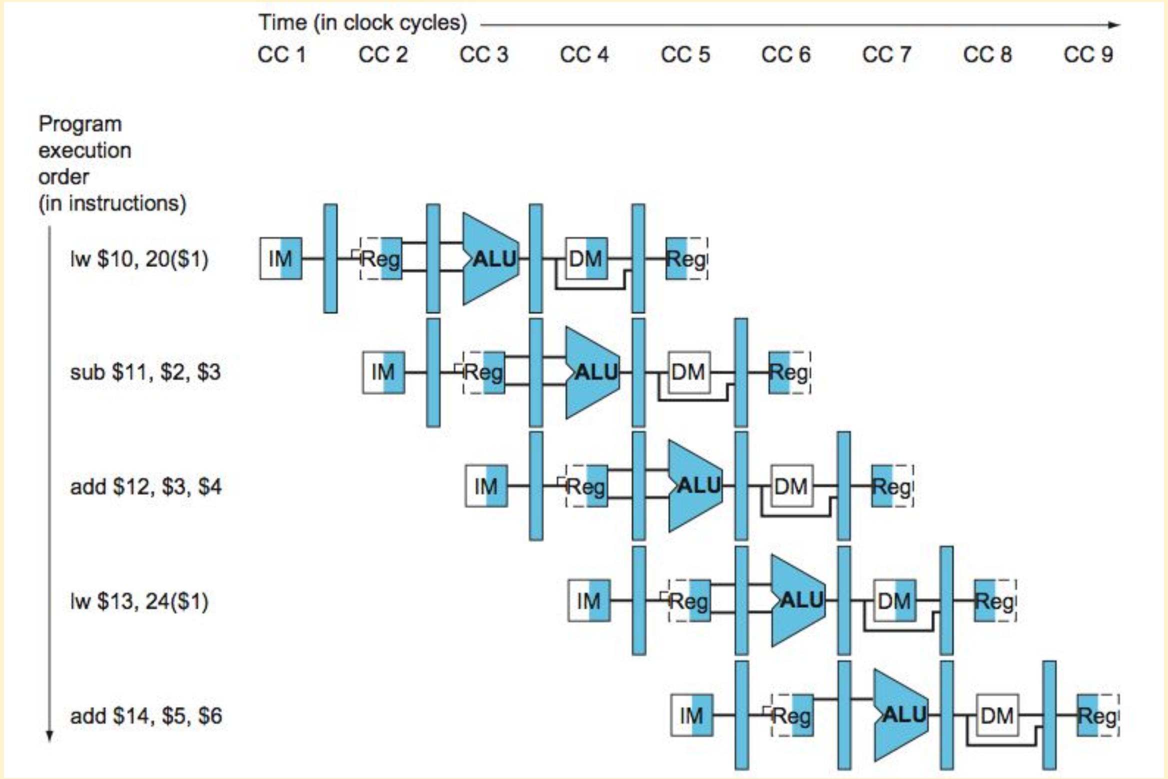
Therefore, we need to carry that register address forward through the pipeline registers.



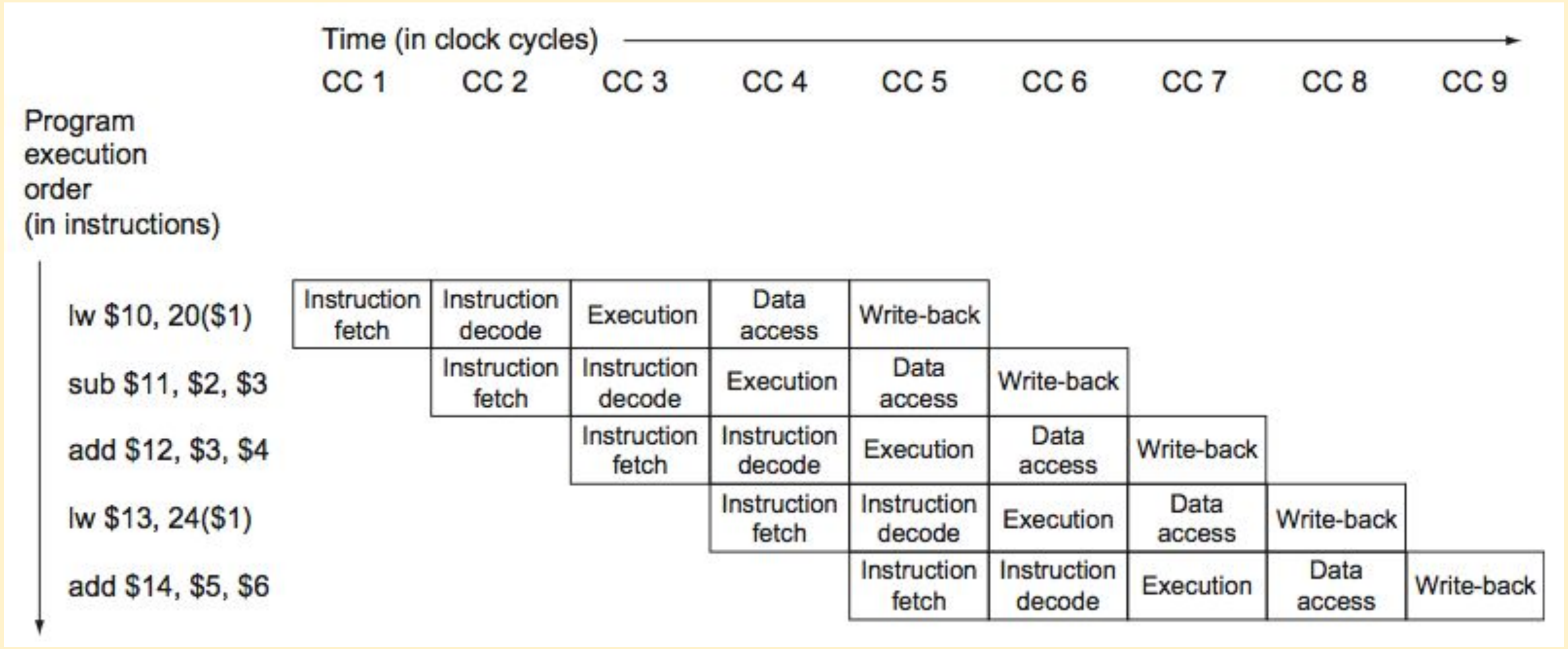
Concurrent instruction execution

clock cycles shown left to right

instructions entering the pipeline are shown top down



Traditional diagram of the same instructions



Time (in clock cycles) →

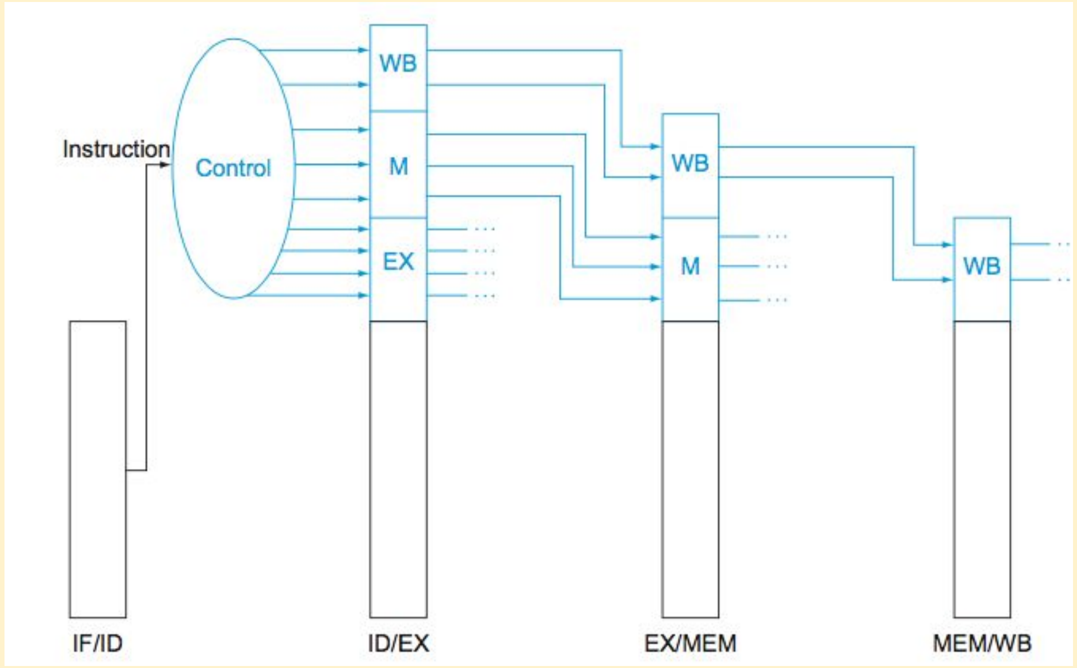
Program execution order (in instructions) ↓

	CC 1	CC 2	CC 3	CC 4	CC 5	CC 6	CC 7	CC 8	CC 9
lw $10, 20($1)	Instruction fetch	Instruction decode	Execution	Data access	Write-back				
sub $11, $2, $3		Instruction fetch	Instruction decode	Execution	Data access	Write-back			
add $12, $3, $4			Instruction fetch	Instruction decode	Execution	Data access	Write-back		
lw $13, 24($1)				Instruction fetch	Instruction decode	Execution	Data access	Write-back	
add $14, $5, $6					Instruction fetch	Instruction decode	Execution	Data access	Write-back

How is control affected by pipelining?

PC and pipeline registers are written every clock cycle, so no control lines are needed.

We will group the control signals by stage so that the control signals for an instruction are carried forward as long as needed.



Control signals grouped by stage

IF and ID stages - no special control signals needed

EX stage - RegDst, ALUOp, and ALUSrc

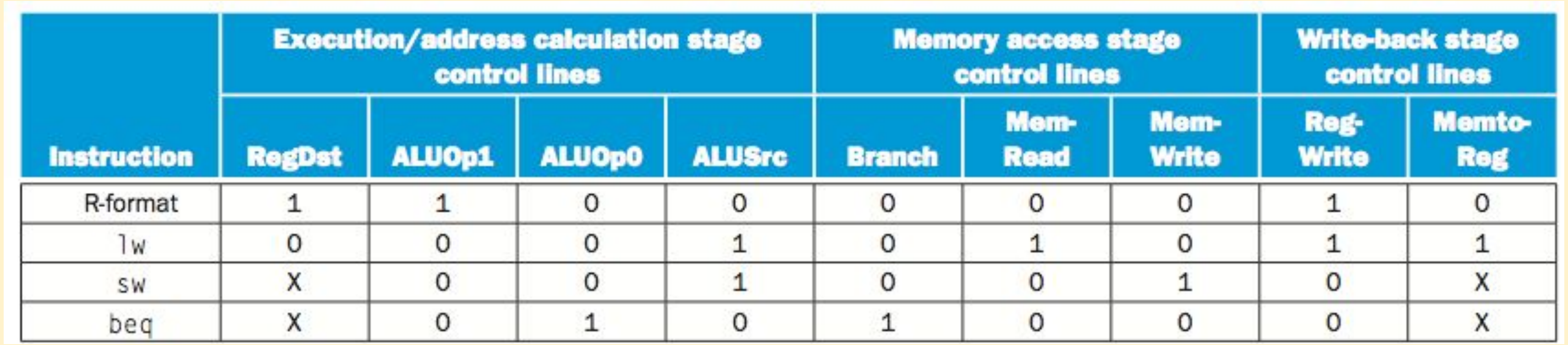
MEM stage - Branch, MemRead, MemWrite

WB stage - MemtoReg, and RegWrite

Control signals have the same meaning as non-pipelined datapath.

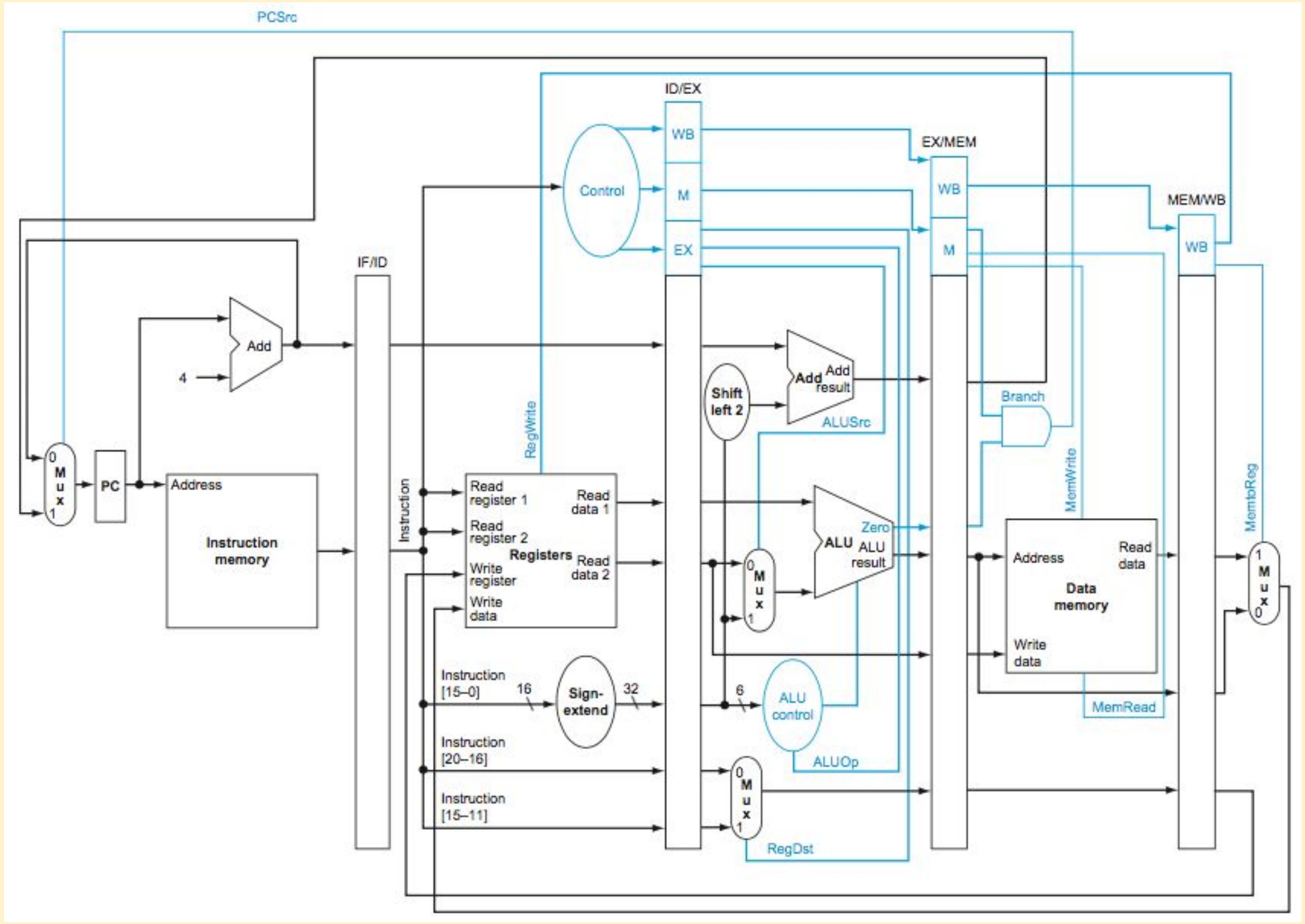
Pipelined control signals

Values are unchanged from non-pipelined datapath, but they are grouped.



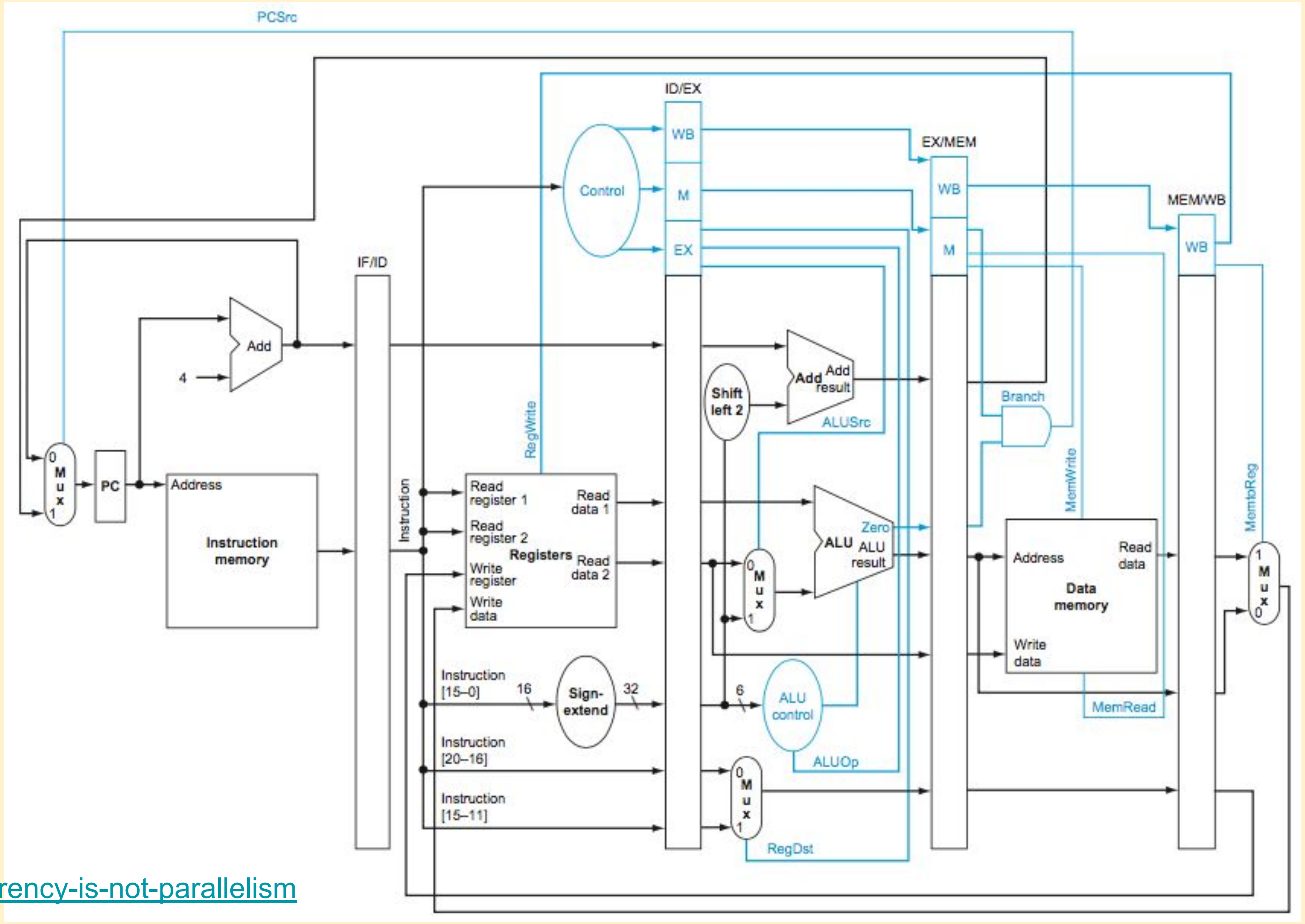
Instruction	Execution/address calculation stage control lines				Memory access stage control lines			Write-back stage control lines	
	RegDst	ALUOp1	ALUOp0	ALUSrc	Branch	Mem-Read	Mem-Write	Reg-Write	Memto-Reg
R-format	1	1	0	0	0	0	0	1	0
lw	0	0	0	1	0	1	0	1	1
sw	X	0	0	1	0	0	1	0	X
beq	X	0	1	0	1	0	0	0	X

Pipeline registers are extended to hold control signals that are carried forward for each instruction.



We can work on up to 5 instructions at the same time within a single CPU core.

Terms concurrent and parallel apply to hardware or software. When to use each term is confusing.



Talk by Rob Pike:

http://blog.golang.org/concurrency-is-not-parallelism

Pipelining at Chipotle