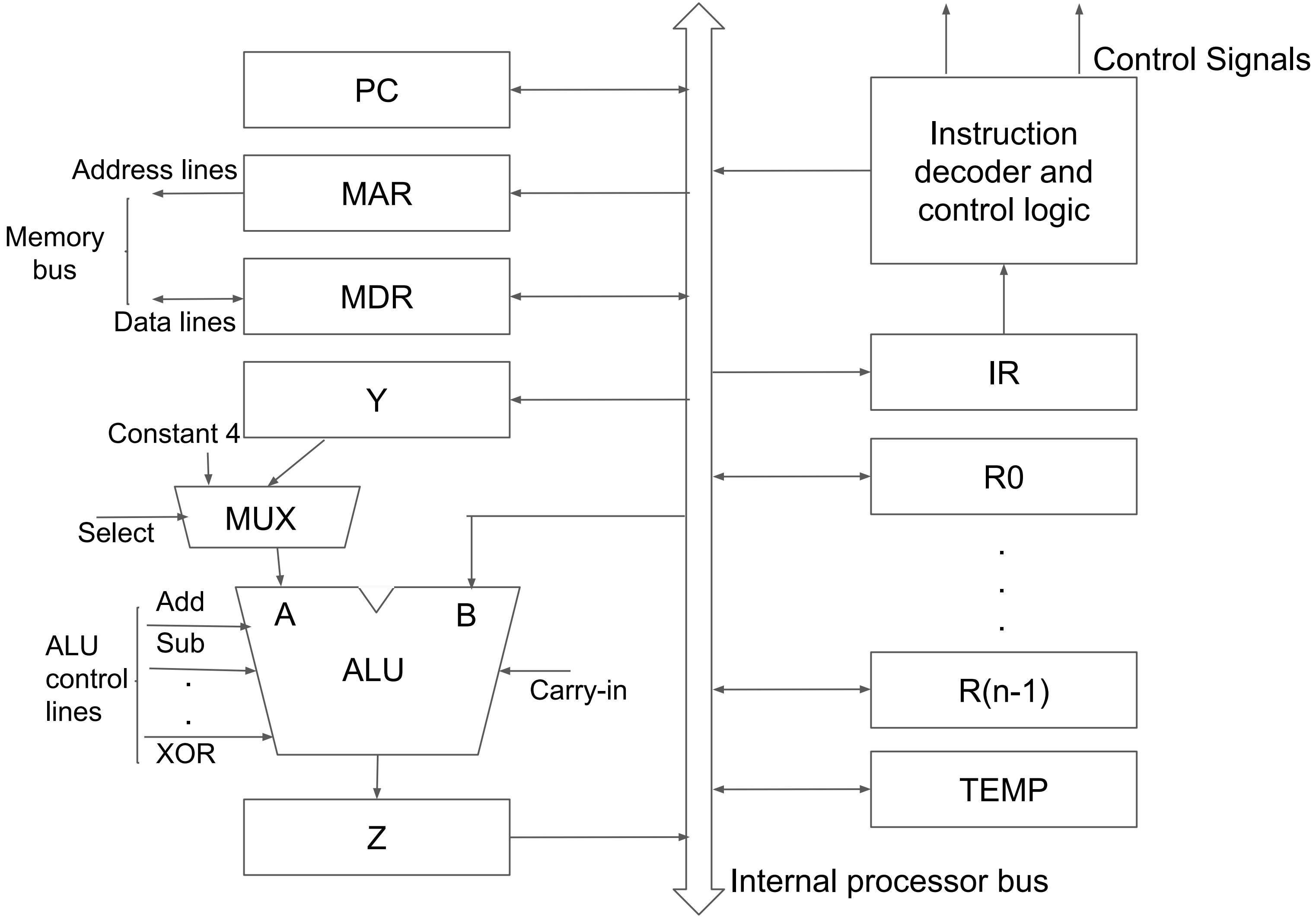
Pipelining

GATE Overflow


PC
Address lines
MAR
Memory bus
MDR
Data lines
Y
Constant 4
MUX
Select
Add
Sub
.
.
XOR
ALU control lines
A
B
ALU
Carry-in
Z
Control Signals
Instruction decoder and control logic
IR
R0
.
.
.
R(n-1)
TEMP
Internal processor bus

Single bus Data Execution

Consider the Execution of the instruction R1 <- R0 + R1

Instruction Fetch (Originally the instruction is in Physical Memory - its address is in PC (this address is Virtual Address but that part is handled by MMU)

Step 1: MOV PC to MAR (Memory Address Register)

Step 2: Get Instruction in MDR (PC increment happens in parallel)

Step 3: MOV MDR to IR

Single bus Data Execution

Consider the Execution of the instruction R1 <- R0 + R1

Instruction Fetch (Originally the instruction is in Physical Memory - its address is in PC (this address is Virtual Address but that part is handled by MMU)

Step 1: PCout, MARin, READ, SELECT 4, ADD, Zin (PC increment also starts here)

Step 2: Zout, PCin, WMFC (Wait for Memory Function Complete), Yin (Yin is only useful for branch instructions to get the target address)

Step 3: MDRout, IRin

Single bus Data Execution

Consider the Execution of the instruction R1 <- R0 + R1

- R0out, Yin
- R1out, ADD, SELECT Y, Zin
- Zout, R0in

Pipelining

Taking some of the slides from Washington University course. For full slides you can visit https://courses.cs.washington.edu/courses/cse378/07au/lectures/L11-Pipelined-Datapath-And.pdf

Pipelining concepts

- A pipelined processor allows multiple instructions to execute at once, and each instruction uses a different functional unit in the datapath.
- This increases throughput, so programs can run faster.
  - One instruction can finish executing on every clock cycle, and simpler stages also lead to shorter cycle times.

	Clock cycle								
	1	2	3	4	5	6	7	8	9
lw $t0, 4($sp)	IF	ID	EX	MEM	WB				
sub $v0, $a0, $a1		IF	ID	EX	MEM	WB			
and $t1, $t2, $t3			IF	ID	EX	MEM	WB		
or $s0, $s1, $s2				IF	ID	EX	MEM	WB	
add $t5, $t6, $0					IF	ID	EX	MEM	WB

Pipelined Datapath

- The whole point of pipelining is to allow multiple instructions to execute at the same time.
- We may need to perform several operations in the same cycle.
  - Increment the PC and add registers at the same time.
  - Fetch one instruction while another one reads or writes data.

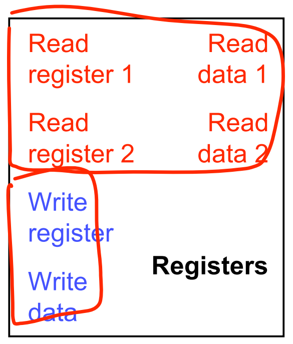
Clock cycle

	1	2	3	4	5	6	7	8	9
lw $t0, 4($sp)	IF	ID	EX	MEM	WB				
sub $v0, $a0, $a1		IF	ID	EX	MEM	WB			
and $t1, $t2, $t3			IF	ID	EX	MEM	WB		
or $s0, $s1, $s2				IF	ID	EX	MEM	WB	
add $t5, $t6, $0					IF	ID	EX	MEM	WB

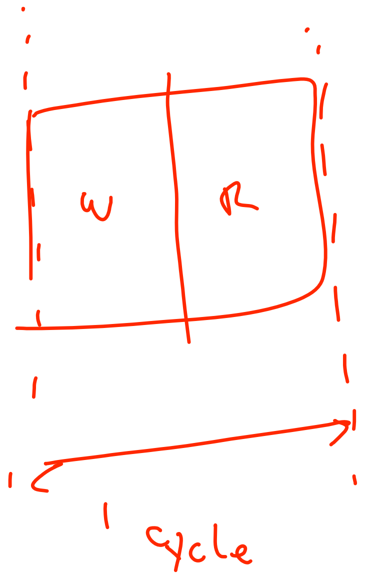
- Thus, like the single-cycle datapath, a pipelined processor will need to duplicate hardware elements that are needed several times in the same clock cycle.

One register file is enough

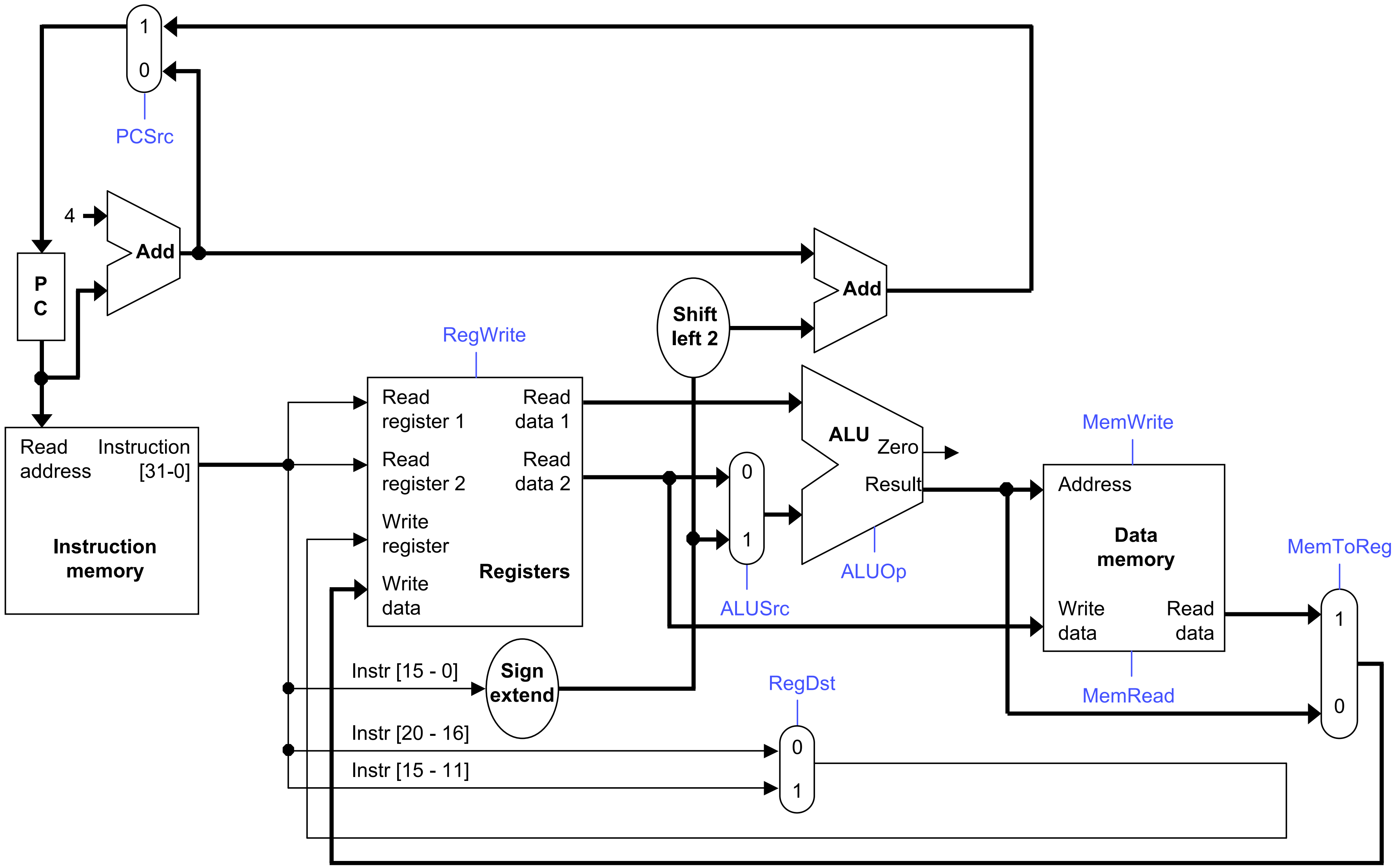
- We need only one register file to support both the ID and WB stages.



- Reads and writes go to separate ports on the register file.
- Writes occur in the first half of the cycle, reads occur in the second half.



Single-cycle datapath, slightly rearranged



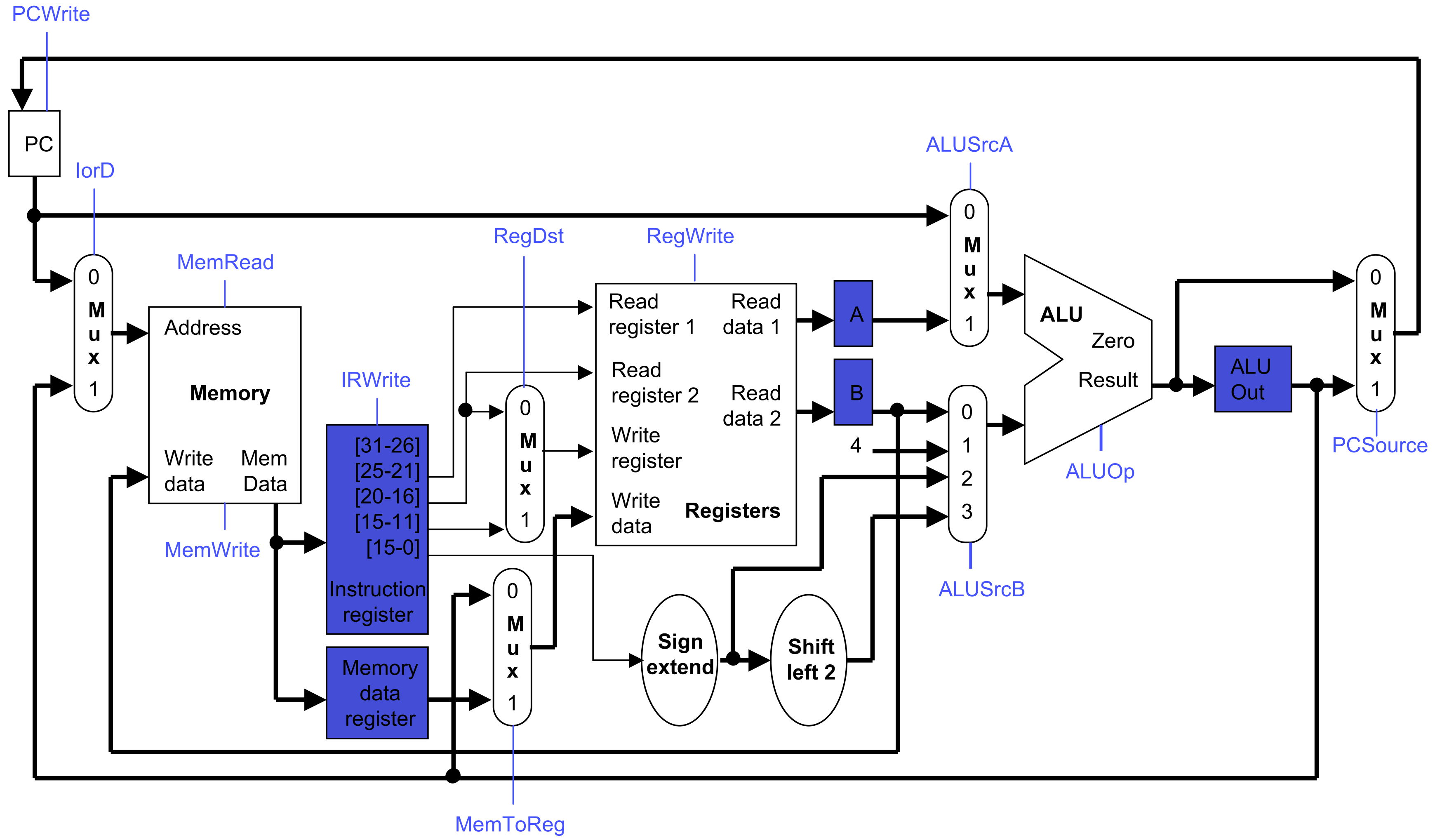
What’s been changed?

- Almost nothing! This is equivalent to the original single-cycle datapath.
  - There are separate memories for instructions and data.
  - There are two adders for PC-based computations and one ALU.
  - The control signals are the same.
- Only some cosmetic changes were made to make the diagram smaller.
  - A few labels are missing, and the muxes are smaller.
  - The data memory has only one Address input. The actual memory operation can be determined from the MemRead and MemWrite control signals.
- The datapath components have also been moved around in preparation for adding pipeline registers.

Multiple cycles

- In pipelining, we also divide instruction execution into multiple cycles.
- Information computed during one cycle may be needed in a later cycle.
  - The instruction read in the IF stage determines which registers are fetched in the ID stage, what constant is used for the EX stage, and what the destination register is for WB.
  - The registers read in ID are used in the EX and/or MEM stages.
  - The ALU output produced in the EX stage is an effective address for the MEM stage or a result for the WB stage.
- We added several intermediate registers to the multicycle datapath to preserve information between stages, as highlighted on the next slide.

Registers added to the multi-cycle

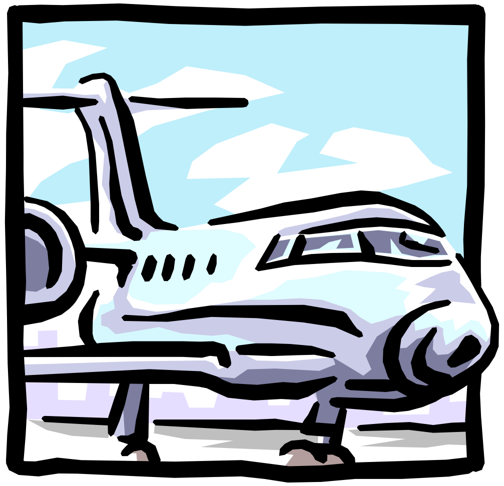


Pipeline registers

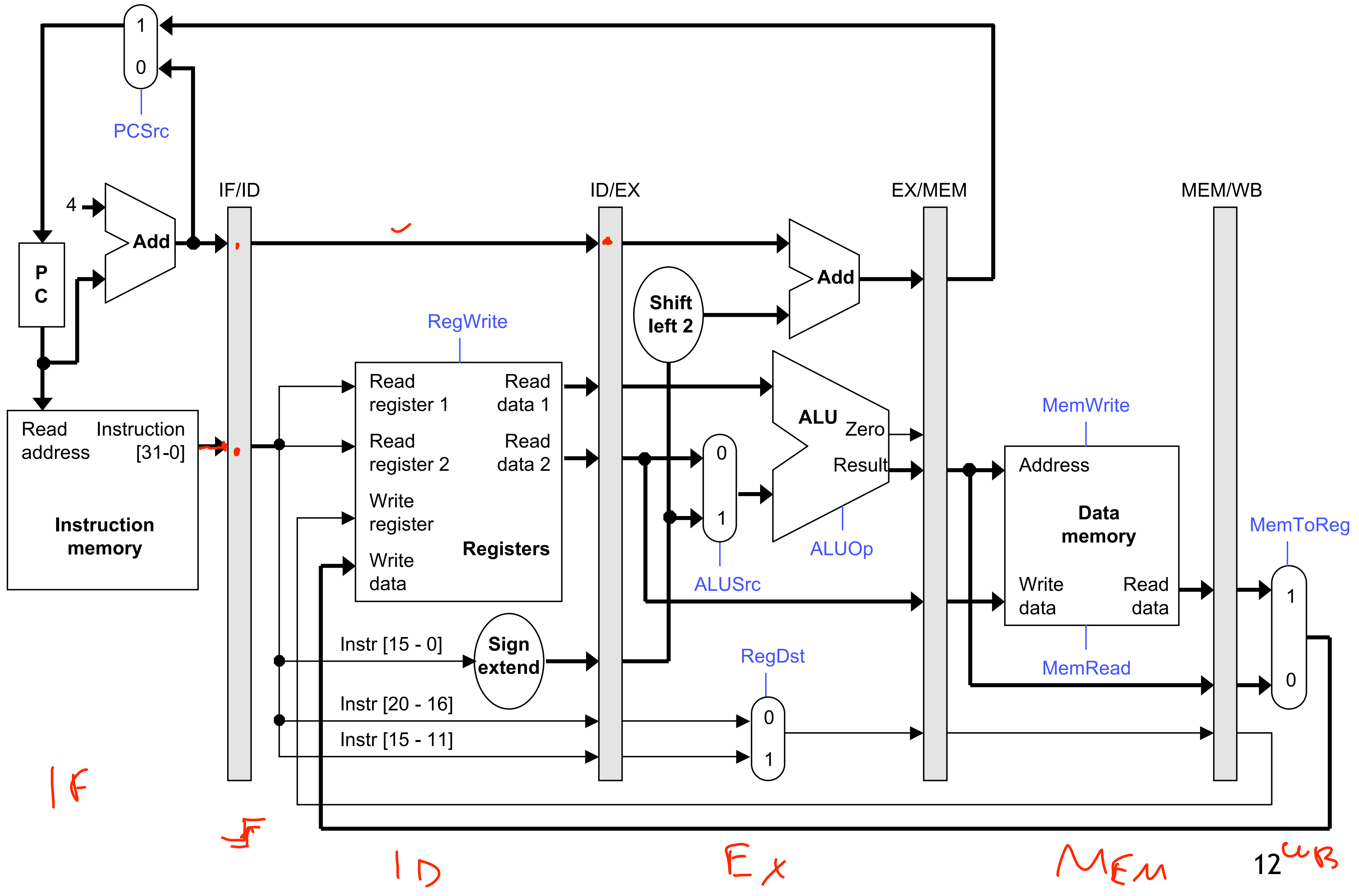
- We'll add intermediate registers to our pipelined datapath too.
- There's a lot of information to save, however. We'll simplify our diagrams by drawing just one big pipeline register between each stage.
- The registers are named for the stages they connect.

IF/ID ID/EX EX/MEM MEM/WB

- No register is needed after the WB stage, because after WB the instruction is done.



Pipelined datapath



That’s a lot of diagrams there

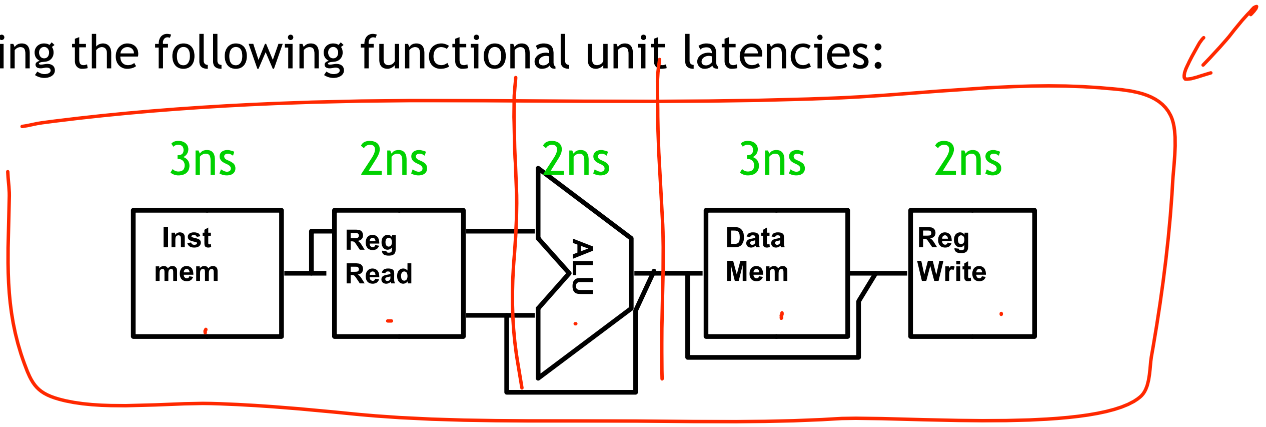
Clock cycle

		1	2	3	4	5	6	7	8	9
lw	$t0, 4($sp)	IF	ID	EX	MEM	WB				
sub	$v0, $a0, $a1		IF	ID	EX	MEM	WB			
and	$t1, $t2, $t3			IF	ID	EX	MEM	WB		
or	$s0, $s1, $s2				IF	ID	EX	MEM	WB	
add	$t5, $t6, $0					IF	ID	EX	MEM	WB

- Compare the last nine slides with the pipeline diagram above.
  - You can see how instruction executions are overlapped.
  - Each functional unit is used by a *different* instruction in each cycle.
  - The pipeline registers save control and data values generated in previous clock cycles for later use.
  - When the pipeline is full in clock cycle 5, all of the hardware units are utilized. This is the ideal situation, and what makes pipelined processors so fast.
- Try to understand this example or the similar one in the book at the end of Section 6.3.

Performance Revisited

- Assuming the following functional unit latencies:



- What is the cycle time of a single-cycle implementation?
  - What is its throughput?

12ns

$\frac{1}{12ns}$

- What is the cycle time of a ideal pipelined implementation?
  - What is its steady-state throughput?

3ns

- How much faster is pipelining?

$\frac{12ns}{3ns} = 4x$

! Speedup

Ideal speedup

Clock cycle

	1	2	3	4	5	6	7	8	9
lw $t0, 4($sp)	IF	ID	EX	MEM	WB				
sub $v0, $a0, $a1		IF	ID	EX	MEM	WB			
and $t1, $t2, $t3			IF	ID	EX	MEM	WB		
or $s0, $s1, $s2				IF	ID	EX	MEM	WB	
add $sp, $sp, -4					IF	ID	EX	MEM	WB

- In our pipeline, we can execute up to five instructions simultaneously.
  - This implies that the maximum speedup is 5 times.
  - In general, the ideal speedup equals the pipeline depth.
- Why was our speedup on the previous slide “only” 4 times?
  - The pipeline stages are imbalanced: a register file and ALU operations can be done in 2ns, but we must stretch that out to 3ns to keep the ID, EX, and WB stages synchronized with IF and MEM.
  - Balancing the stages is one of the many hard parts in designing a pipelined processor.

The pipelining paradox

Clock cycle

		1	2	3	4	5	6	7	8	9
lw	$t0, 4($sp)	IF	ID	EX	MEM	WB				
sub	$v0, $a0, $a1		IF	ID	EX	MEM	WB			
and	$t1, $t2, $t3			IF	ID	EX	MEM	WB		
or	$s0, $s1, $s2				IF	ID	EX	MEM	WB	
add	$sp, $sp, -4					IF	ID	EX	MEM	WB

- Pipelining does *not* improve the execution time of any single instruction. Each instruction here actually takes *longer* to execute than in a single-cycle datapath (15ns vs. 12ns)!
- Instead, pipelining increases the throughput, or the amount of work done per unit time. Here, several instructions are executed together in each clock cycle.
- The result is improved execution time for a *sequence* of instructions, such as an entire program.