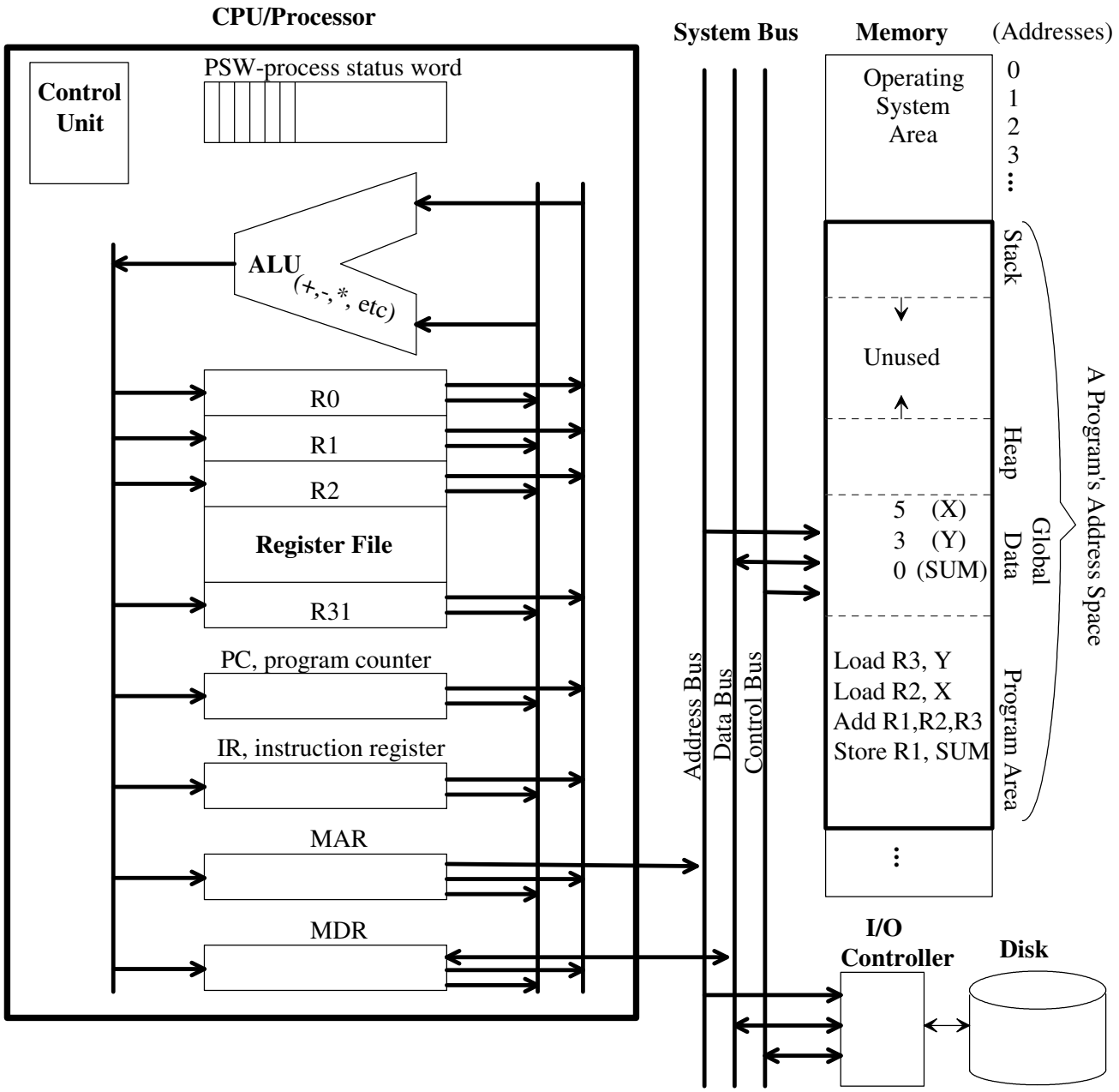
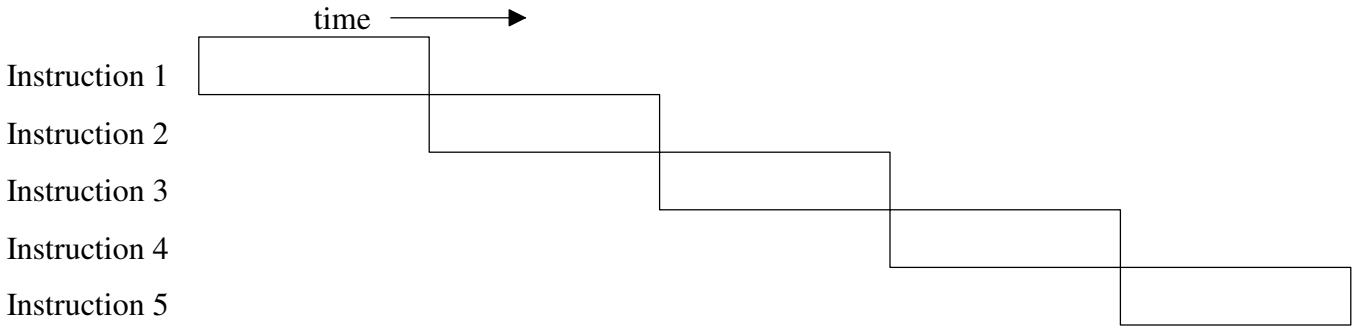


Instruction/Machine Cycle of stored-program computer - repeat all day

1. Fetch Instruction - read instruction pointed at by the program counter (PC) from memory into Instr. Reg. (IR)
2. Decode Instruction - figure out what kind of instruction was read
3. Fetch Operands - get operand values from the memory or registers
4. Execute Instruction - do some operation with the operands to get some result
5. Write Result - put the result into a register or in a memory location

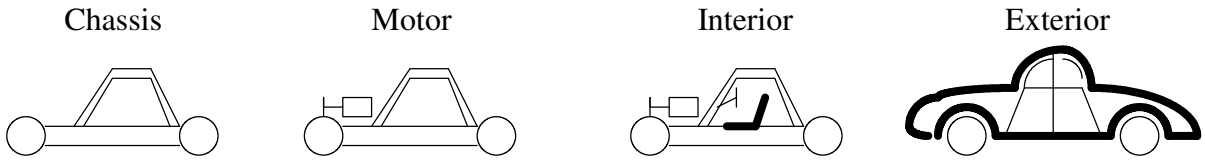
Note: Sometime during the above steps, the PC is updated to point to the next instruction.

Serial Execution - complete fetch-decode-execute cycle before starting the next instruction



Instruction Pipelining - assembly-line idea used to speed instruction completion rate

Assume that an automobile assembly process takes 4 hours.

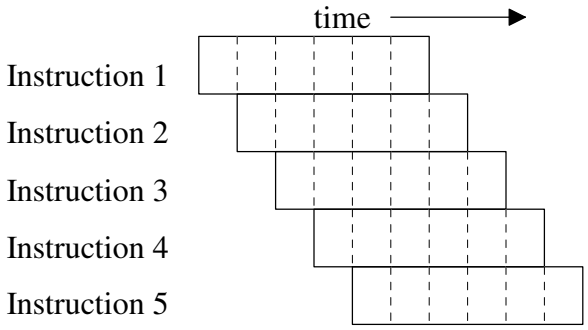


If you divide the process into four equal stages, then ideally

$$\text{time between completions} = \frac{\text{time to complete one car}}{\text{\# of stages}}$$

Problems:
- stages might not be balanced
- overhead of moving cars between stages
- two stages need same specialized tool (structural hazard)

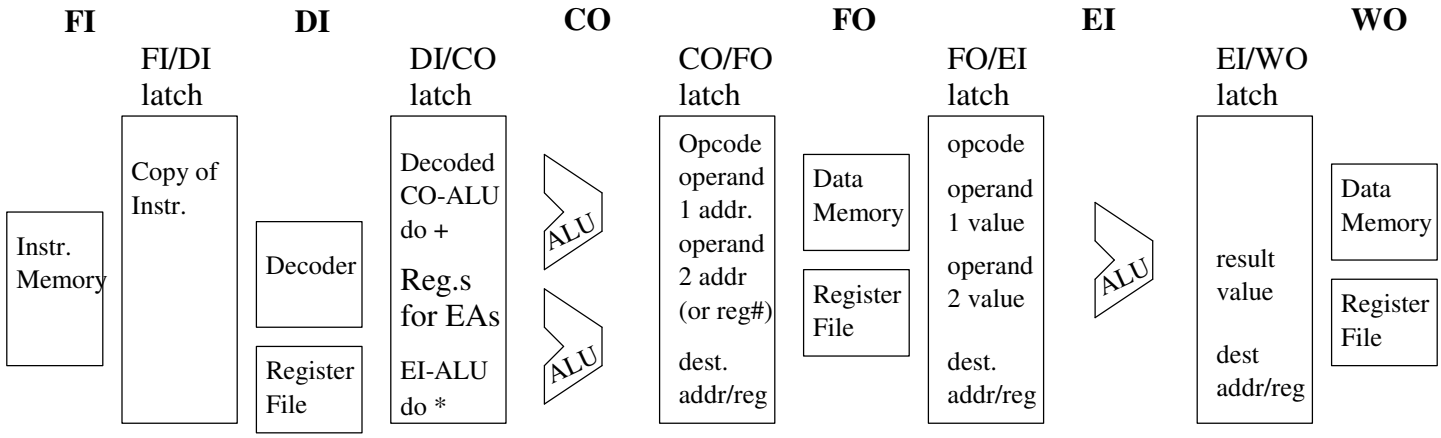
Pipelined Execution - goal is to complete one instruction per clock cycle



Instruction Pipelining Example: One possible break down of instruction execution.

Stage	Abbreviation	Actions
Fetch Instruction	FI	Read next instruction into CPU
Decode Instruction	DI	Determine opcode and operand specifiers; Read registers involved in operand address calculations performed in the CO stage
Calculate Operands	CO	Calculate the effective addresses of all operands
Fetch Operands	FO	Fetch operands from memory or register file
Execute Instruction	EI	Perform the indicated operation
Write Operand	WO	Write operand to memory or register file

Pipeline latches/registers between each stage. Hold temporary results and act like an IR. Some of the hardware components (e.g., Memory and Register File) are shown as if they are duplicated, but they are not.



Problems that delay/*stall* the pipeline:

- ***structural hazard -*** a piece of hardware is needed by several stages at the same time, e.g., Memory in FI, FO, and WO. This might require stages to sequentially access the hardware.
- ***data hazard*** - an instruction depends on the results of a previous instruction which has not been calculated yet. (RAW) read-after-write example:

  ADD R3, R2, R1 ; R3← R2 + R1
  SUB R4, R3, R5 ; R4 ← R3 + R5

  In what stage does the ADD instruction update R3?

  In what stage does the SUB instruction read R3?

- ***control hazard*** - branching makes it difficult to fetch the "correct" instructions to be executed

Data Hazards - the problem

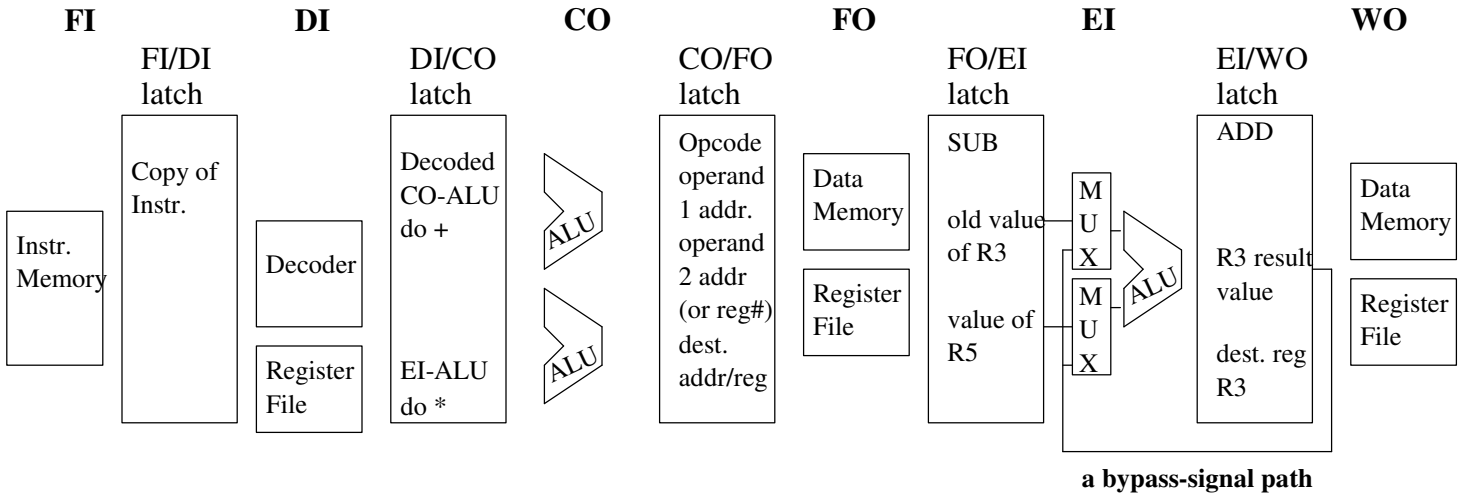
	Time →														
Instructions	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
ADD R3, R2, R1	FI	DI	CO	FO	EI	**WO**									
SUB R4, R3, R5		FI	DI	CO	**FO**	EI	WO								

Solution Alternatives:

1) Introduce stalls

	Time →														
Instructions	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
ADD R3, R2, R1	FI	DI	CO	FO	EI	**WO**									
SUB R4, R3, R5		FI	DI	CO	stall	stall	**FO**	EI	WO						

2) Add additional hardware (***bypass-signal paths***) to "**foward**" R3's new value to the SUB instruction:

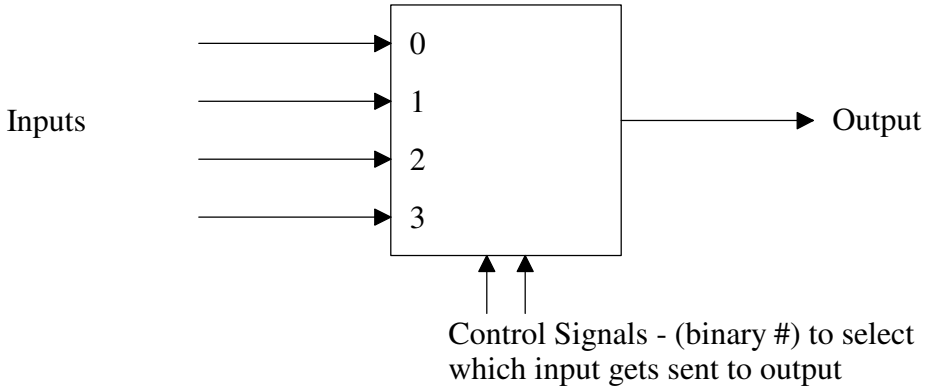


No stalls needed in this case.

	Time →														
Instructions	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
ADD R3, R2, R1	FI	DI	CO	FO	**EI**	WO									
SUB R4, R3, R5		FI	DI	CO	FO	**EI**	WO								

What would control the MUX?

MUX Operation:



Consider the following code: ADD R3, R2, R1
LOAD R4, 4(R3)

What would the timing be **without** bypass-signal paths/forwarding?

	Time →														
Instructions	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
ADD R3, R2, R1	FI	DI	CO	FO	EI	**WO**									
LOAD R4, 4(R3)		FI													

This assumes that R3 cannot be written and the new value read in the same stage.

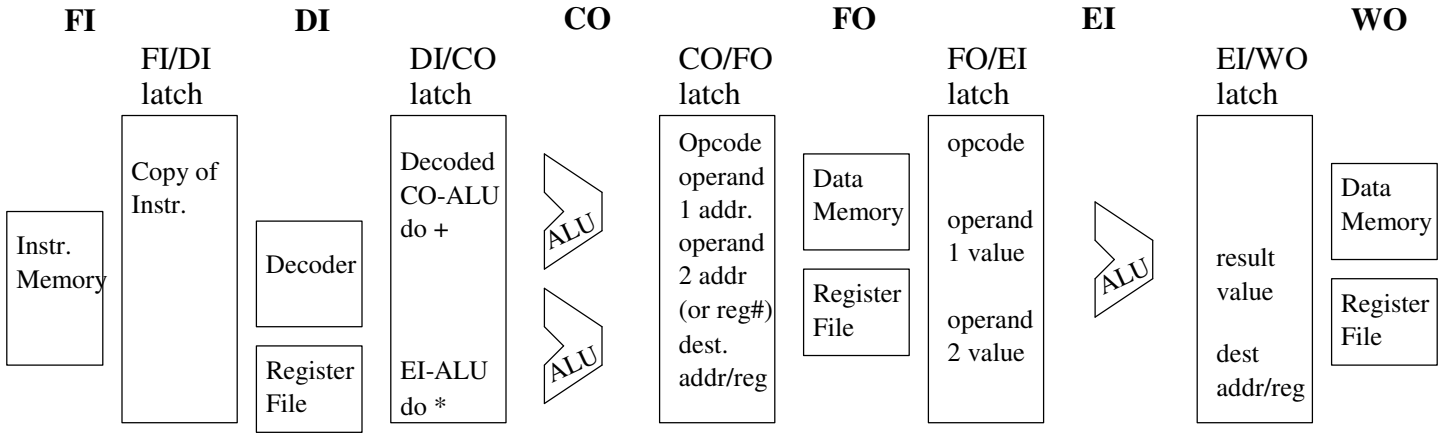
If we assume that R3 can be written in the first half of the WO stage and its new value read in the last half of the DI stage, then we get:

	Time →														
Instructions	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
ADD R3, R2, R1	FI	DI	CO	FO	EI	**WO**									
LOAD R4, 4(R3)		FI													

What would the timing be with bypass-signal paths?

	Time →														
Instructions	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
ADD R3, R2, R1	FI	DI	CO	FO	**EI**	WO									
LOAD R4, 4(R3)		FI													

Draw the bypass-signal paths needed for the above example.



Consider a larger section of code. What would the timing be **without** bypass-signal paths/forwarding (use "stalls" to solve the data hazard)? (This code might require more or less that 15 cycles)

	Time →														
Instructions	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
LOAD R4, 16(R3)	FI	DI	CO	FO	EI	WO									
ADD R3, R2, R1		FI													
STORE R3, 8(R4)															
LOAD R1, 4(R3)															
SUB R6, R1, R8															
ADD R5, R3, R6															

(Assume that a register **cannot** be written and the new value read in the same stage.)

b. What would the timing be **with** bypass-signal paths?
(This code might require more that 15 cycles)

	Time →														
Instructions	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
LOAD R4, 16(R3)	FI	DI	CO	FO	EI	WO									
ADD R3, R2, R1		FI													
STORE R3, 8(R4)															
LOAD R1, 4(R3)															
SUB R6, R1, R8															
ADD R5, R3, R6															

(Assume that a register **cannot** be written and the new value read in the same stage.)

c. Draw ALL the bypass-signal paths needed for the above example.