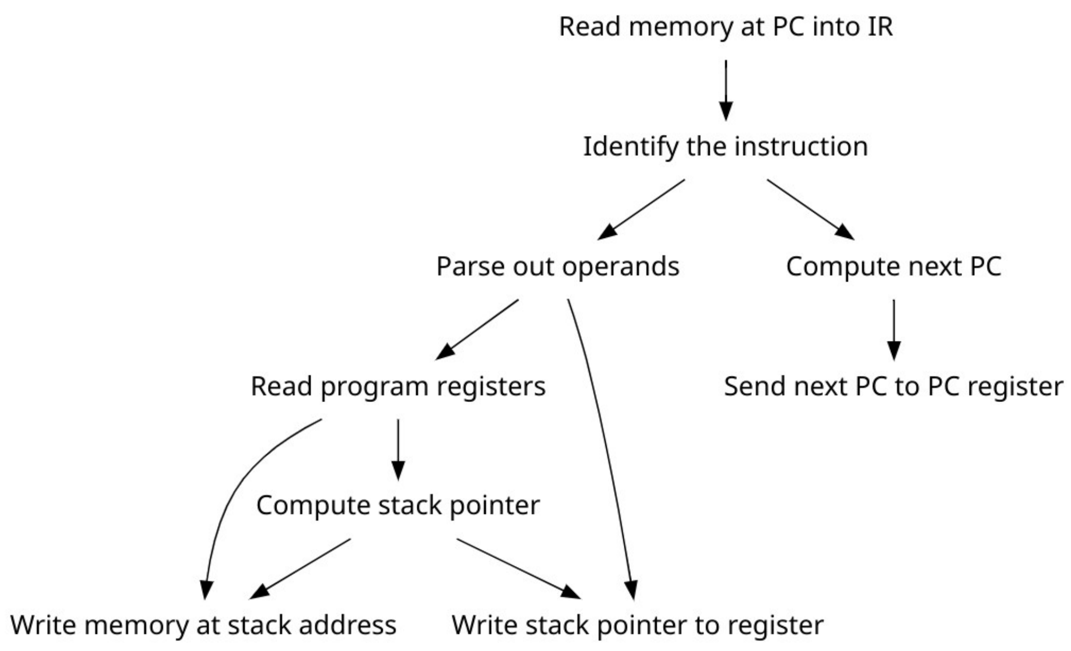
# CPU instructions

## Pipelining and beyond

Executing most instructions involved several steps, some of which depend on others. For example, the `push` instruction involves the following:


Read memory at PC into IR
Identify the instruction
Parse out operands
Compute next PC
Read program registers
Send next PC to PC register
Compute stack pointer
Write memory at stack address
Write stack pointer to register

While each instruction has its own dependency graph of steps, they can all be fit into the following sequence:

1. Fetch an instruction from memory
2. Decode what the instruction intends to do
3. Retrieve values from the register file
4. Execute any math or logic operations the instruction requires
5. Access memory
6. Update values in the register file

### 5.6.3. Footnotes

1. One suggestion is "Computer Architecture: A Quantitative Approach", by John Hennessy and David Patterson.

## 5.7. Pipelining: Making the CPU Faster

Our four-stage CPU takes four cycles to execute one instruction: the first cycle is used to fetch the instruction from memory; the second to decode the instruction and read operands from the register file; the third for the ALU to execute the operation; and the fourth to write back the ALU result to a register in the register file. To execute a sequence of *N* instructions takes *4N* clock cycles, as each is executed one at a time, in order, by the CPU.

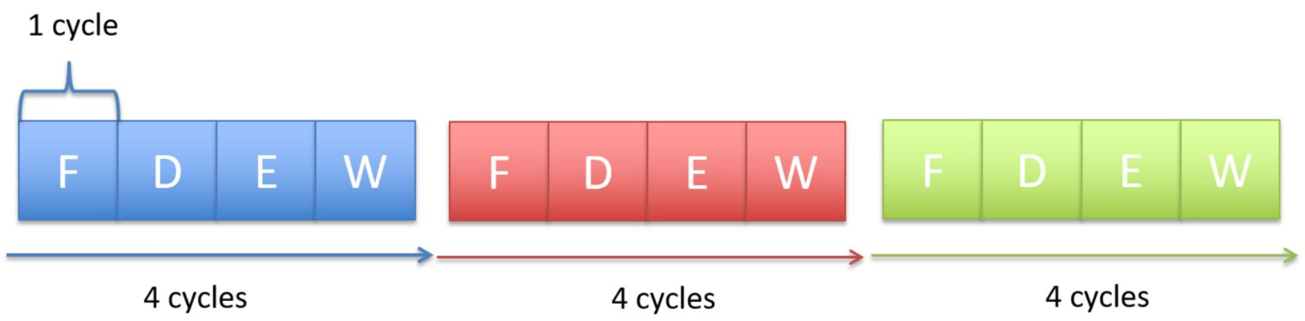
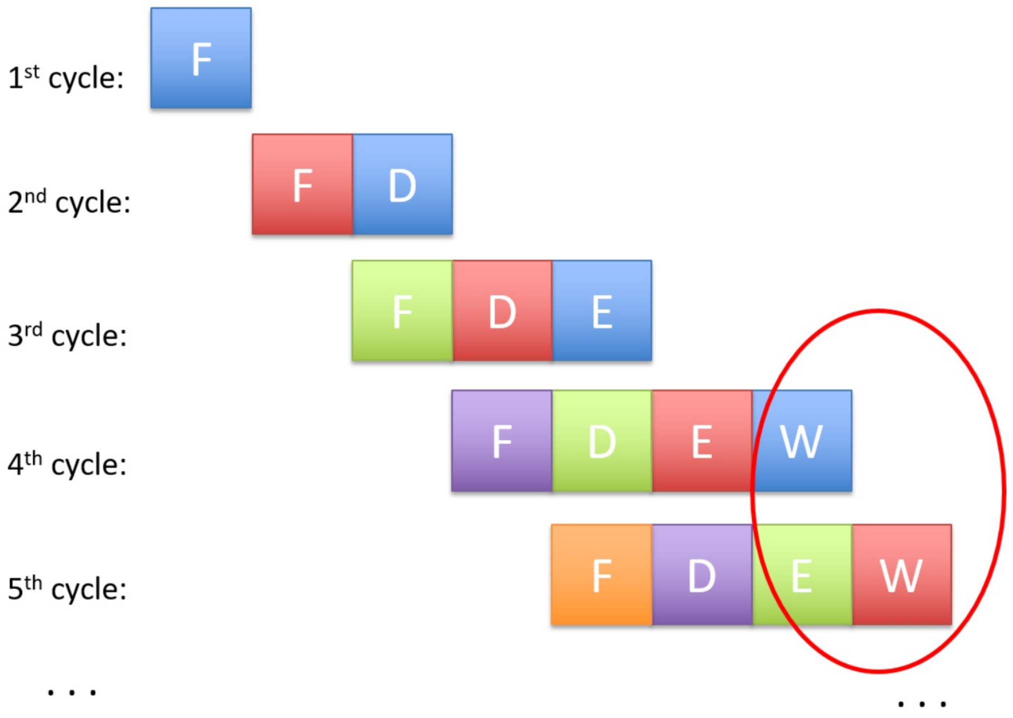


*Figure 84. Executing three instructions takes 12 total cycles.*

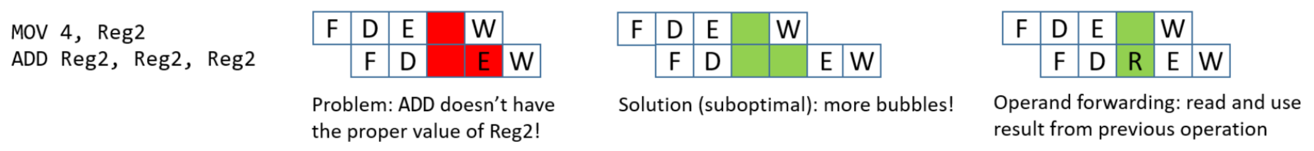


- **Fetch (F):** reads an instruction from memory (pointed to by the program counter).
- **Decode (D):** reads source registers and sets control logic.
- **Execute (E):** executes the instruction.
- **Memory (M):** reads from or writes to data memory.
- **WriteBack (W):** stores a result in a destination register.

The processor prevents the aforementioned scenario by first forcing every instruction to take five pipeline stages to execute. For instructions that normally take fewer than five stages, the CPU adds a "no-operation" ( `NOP` ) instruction (also called a pipeline "bubble") to substitute for that phase.

However, the problem is still not fully resolved. Since the goal of the second instruction is to add `2` to the value stored in register `Reg1` , the `MOV` instruction needs to finish *writing* to register `Reg1` before the `ADD` instruction can execute correctly. A similar problem exists in the next two instructions:

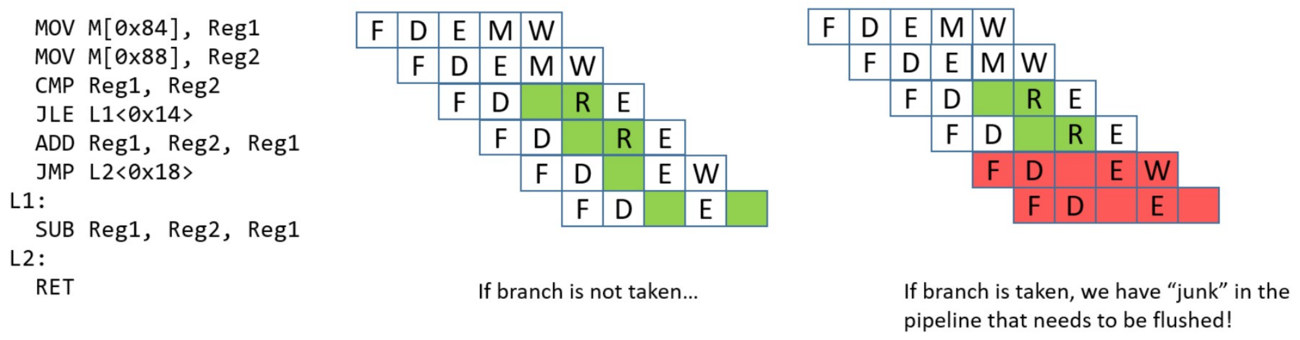
```
MOV 4, Reg2            # copy the value 4 to register Reg2
ADD Reg2, Reg2, Reg2   # compute Reg2 + Reg2, store result in Reg2
```



```
MOV 4, Reg2
ADD Reg2, Reg2, Reg2
```

Problem: ADD doesn't have the proper value of Reg2!
Solution (suboptimal): more bubbles!
Operand forwarding: read and use result from previous operation

*Figure 87. The processor can reduce the damage caused by pipeline hazards by forwarding operands between instructions.*

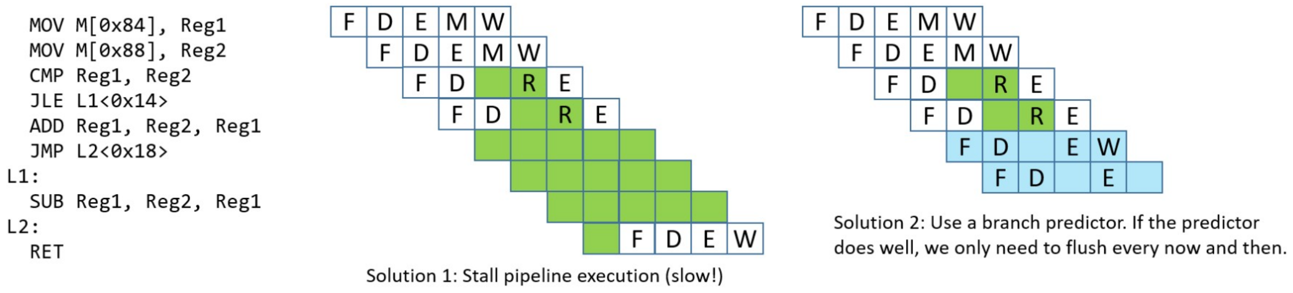


```
  MOV M[0x84], Reg1
  MOV M[0x88], Reg2
  CMP Reg1, Reg2
  JLE L1<0x14>
  ADD Reg1, Reg2, Reg1
  JMP L2<0x18>
L1:
  SUB Reg1, Reg2, Reg1
L2:
  RET
```

If branch is not taken...
If branch is taken, we have "junk" in the pipeline that needs to be flushed!

*Figure 88. An example of a control hazard resulting from a conditional branch.*

dereferences and side effects.



```
  MOV M[0x84], Reg1
  MOV M[0x88], Reg2
  CMP Reg1, Reg2
  JLE L1<0x14>
  ADD Reg1, Reg2, Reg1
  JMP L2<0x18>
L1:
  SUB Reg1, Reg2, Reg1
L2:
  RET
```

Solution 1: Stall pipeline execution (slow!)

Solution 2: Use a branch predictor. If the predictor does well, we only need to flush every now and then.

*Figure 89. Potential solutions for handling control hazards.*