SEC204

Computer Architecture and Low Level Programming

Dr. Vasilios Kelefouras

Email: v.kelefouras@plymouth.ac.uk

Website:

https://www.plymouth.ac.uk/staff/vasilios-kelefouras

Instruction Pipeline

- So far we have used the following sequence: fetch instruction, decode instruction, execute instruction
  - Notice that each execution step uses a different functional unit
  - But the units are idle most of the time
    - That's a lot of hardware sitting around doing nothing
- **We shouldn't have to wait for an entire instruction to complete before we can re-use the function units**
  - **Pipelining** solves the above inefficiency
- Pipelining is a general technique applied in our everyday life, not just to computers, e.g., restaurants

Instruction Pipeline – **an analogy to laundry**

- Assume we have got
  - One washer *(takes 1 hour)* 

  - One Drier *(takes 1 hour)* 

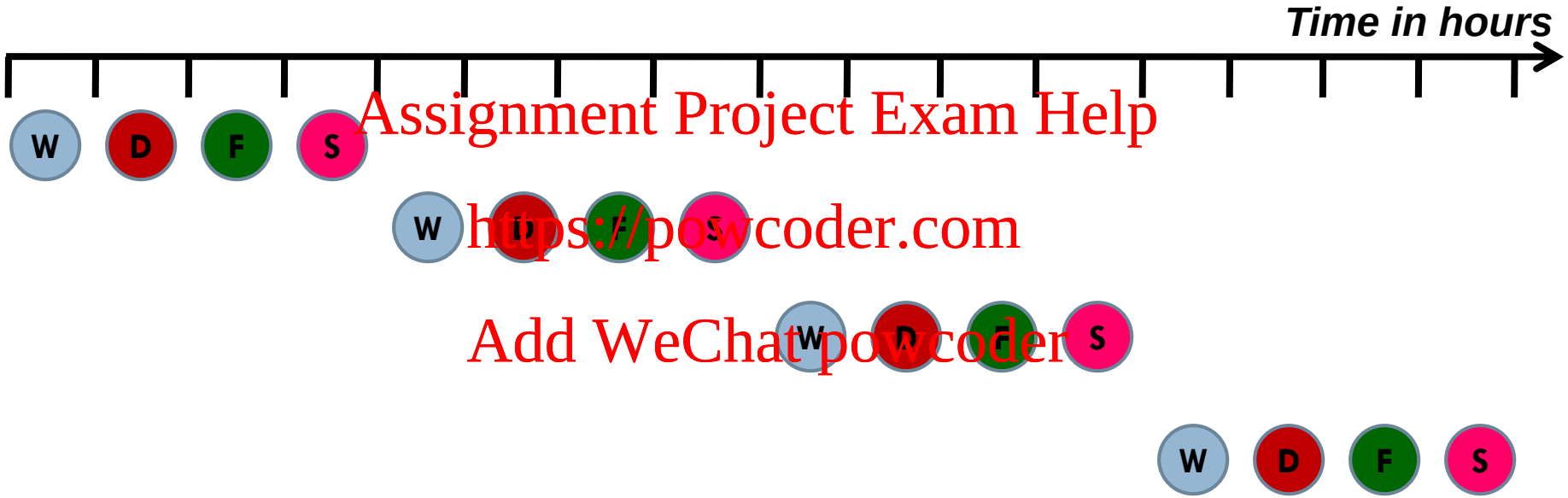
  - One Folder *(takes 1 hour)* 

  - Something/someone to store the clothes *(takes 1 hour)* 

- So, it takes 4 hours to wash, dry, fold and store a load of laundry

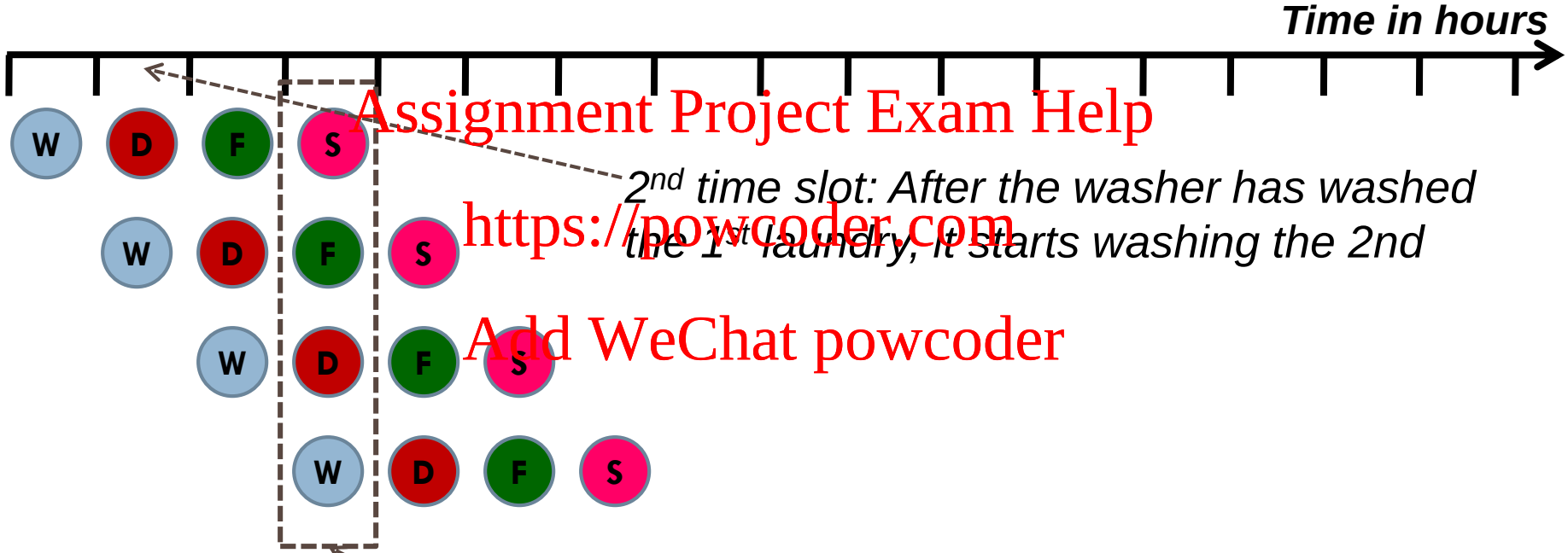
Instruction Pipeline – an analogy to laundry (1)

- Assume we have got 4 loads of laundry – we have to wait for 16 hours



Instruction Pipeline – an analogy to laundry (2)

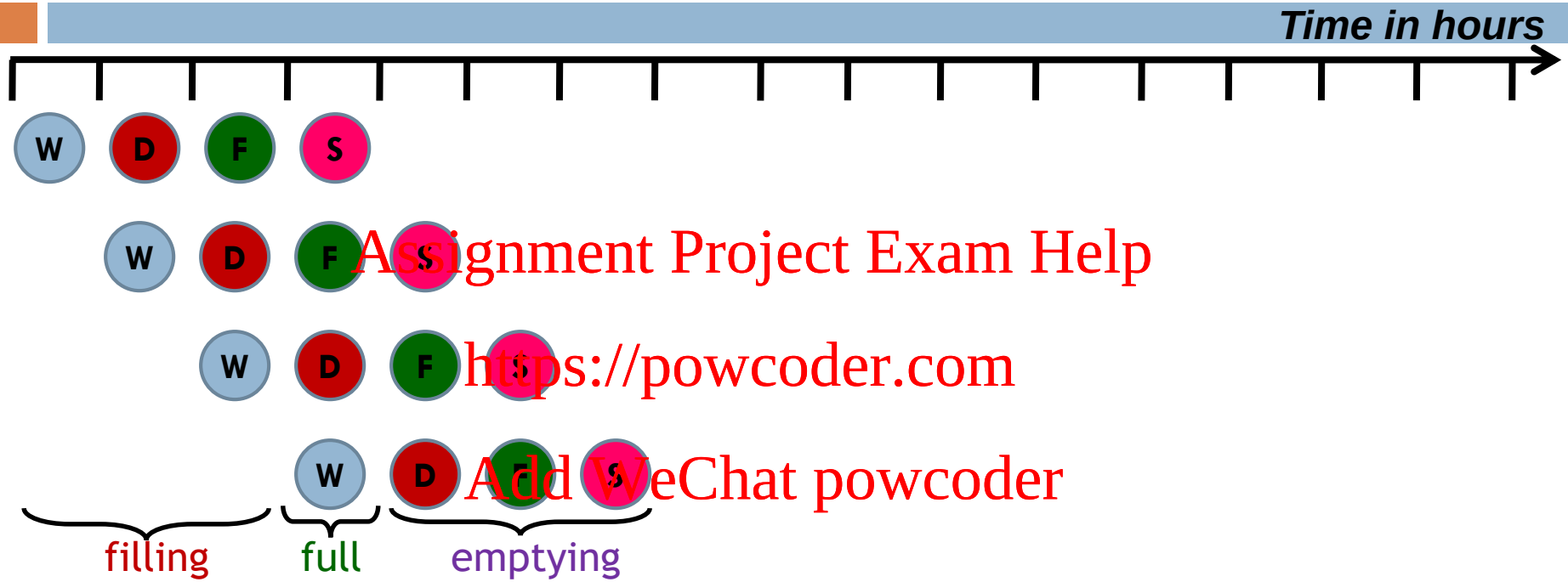
- But, if we use pipelining technique, we need to wait just for 7 hours



2nd time slot: After the washer has washed the 1st laundry, it starts washing the 2nd

4th time slot: The washer washes the 4th load, the drier dries the 3th load, the folder folds the 2nd load and the man stores the 1st load

Instruction Pipeline – an analogy to laundry (3)



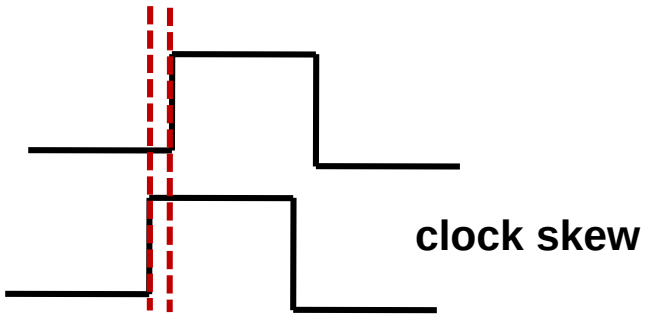
- The **latency** of a single load remains 4 hours
- **But throughput is increased - the number of loads completed per unit of time**
  - Finish the execution of a load after every clock cycle
- The time to fill and drain the pipeline reduces throughput, but it happens only at the beginning and at the end, respectively
  - Consider 1000 laundries
- The maximum speedup equals to the number of pipeline stages

Instruction Pipeline – back to computers (1)

- Now, the time in hours becomes time in **CPU clock cycles**
- Different processors have different number of pipeline stages
  - Many designs include pipelines as long as 7, 10 and even 20 stages
- The instructions are divided into smaller ones which are performed by different processor units
- Each pipeline stage needs one CPU cycle
- **Pipeline increases throughput, but increases latency due to the added overhead of the pipelining process itself** (explain next)
- As the pipeline is made "deeper" (with a greater number of steps), a given step can be implemented with simpler circuitry, which may let the processor clock run faster

Instruction Pipeline – back to computers (2)

- **Pipelining increases the CPU instruction throughput** - the number of instructions completed per unit of time
  - it does not reduce the execution time of an individual instruction
  - In fact, it usually **slightly increases the execution time of each instruction due to overhead in the pipeline control**
    - **Pipeline overhead arises because extra hardware is needed (registers) which introduce a delay for several reasons, e.g., clock skew**
      - Clock skew is a phenomenon in synchronous circuits in which the clock signal arrives at different components at different times
- The increase in instruction throughput means that a program runs faster and has lower total execution time



Instruction Pipeline - Performance Issues (1)

Consider a **non-pipelined** machine with 5 execution stages of lengths 50 ns, 50 ns, 60 ns, 60 ns, and 50 nseconds

1. Find the instruction latency
2. How much time does it take to execute 100 instructions?

Instruction latency = 50+50+60+60+50= 270ns

Time to execute 100 instructions = 100*270 = 27000 ns

Instruction Pipeline - Performance Issues (2)

Suppose we introduce pipelining

Assume that when introducing pipelining, the clock skew adds 5ns of overhead to each execution stage

1. What is the instruction latency on the pipelined machine?
2. How much time does it take to execute 100 instructions?

The length of the pipe stages must all be the same, i.e., the speed of the slowest stage plus the overhead

The length of pipelined stage = max(lengths of unpipelined stages) + overhead = 60 + 5 = 65 ns

Instruction latency = 65 ns

Time to execute 100 instructions = 65*5*1 + 65*1*99 = 325 + 6435 = 6760 ns

Speedup for 100 instructions = 27000 / 6760 = 3.994≈4 (average instruction time without pipelining to the average instruction time with pipelining)

The classic five stage RISC pipeline

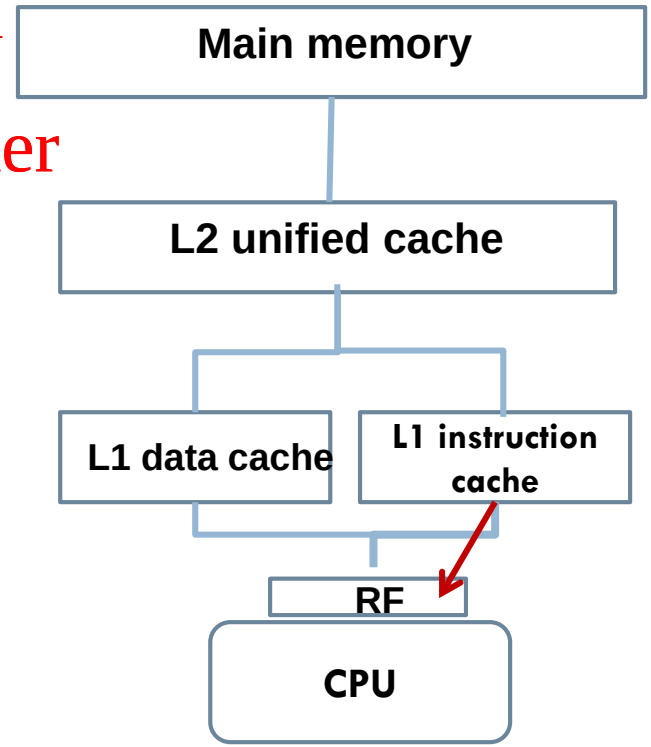
1. **Instruction Fetch:** Read an instruction from memory
2. **Instruction Decode:** identify the instruction and its operands
3. **Execute:** execute an arithmetical instruction or compute the address of a load/store
4. **Memory:** load or store from/to memory
5. **Write Back:** Store the result in the destination register

Note, not all instructions need all five steps

The classic five stage RISC pipeline

Instruction Fetch (IF):

- The instruction is read from memory
- The PC shows the address of the next instruction
- The instruction is loaded from L1 instruction cache (1 cycle)
  - But the next instruction is not always in the instruction cache
- The instruction is stored into the Instruction Register (IR)



The classic five stage RISC pipeline

Instruction Decode (ID)

- The processor reads the Instruction Register (IR) and identifies the instruction
- Reads any operands required from the register file
- The CPU generates the control signals
- The instruction decode phase will calculate the next PC and will send it back to the IF phase so that the IF phase knows which instruction to fetch next

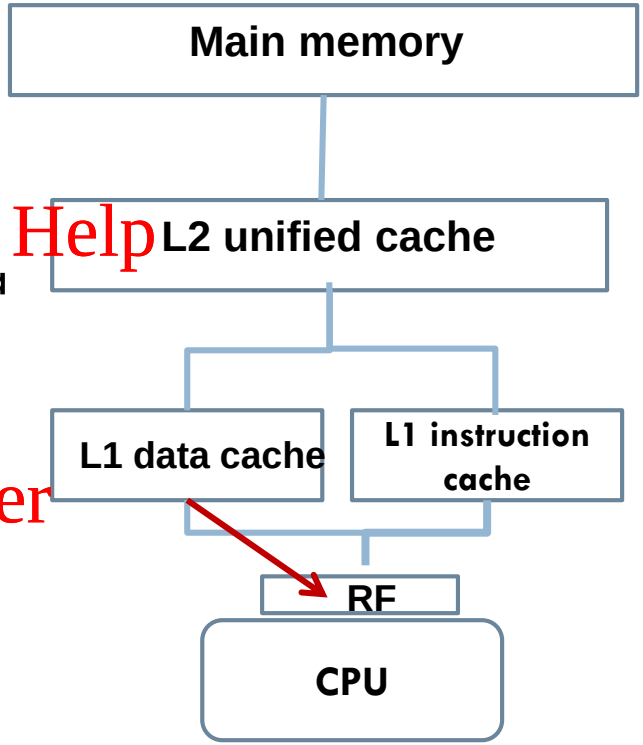
Execute:

- The Execute stage is where the actual computation occurs
- These calculations are all done by the ALU
- The arithmetical instructions are executed at this stage
- For load/store instructions, the address calculation is made

The classic five stage RISC pipeline

Memory:

- The Memory Access stage performs any memory access required by the current instruction
- So, for loads, it would load an operand from L1 data cache memory
- For stores, it would store an operand into memory
- For all other instructions, it would do nothing
- Note that the data are not always in L1 data cache memory



Write Back:

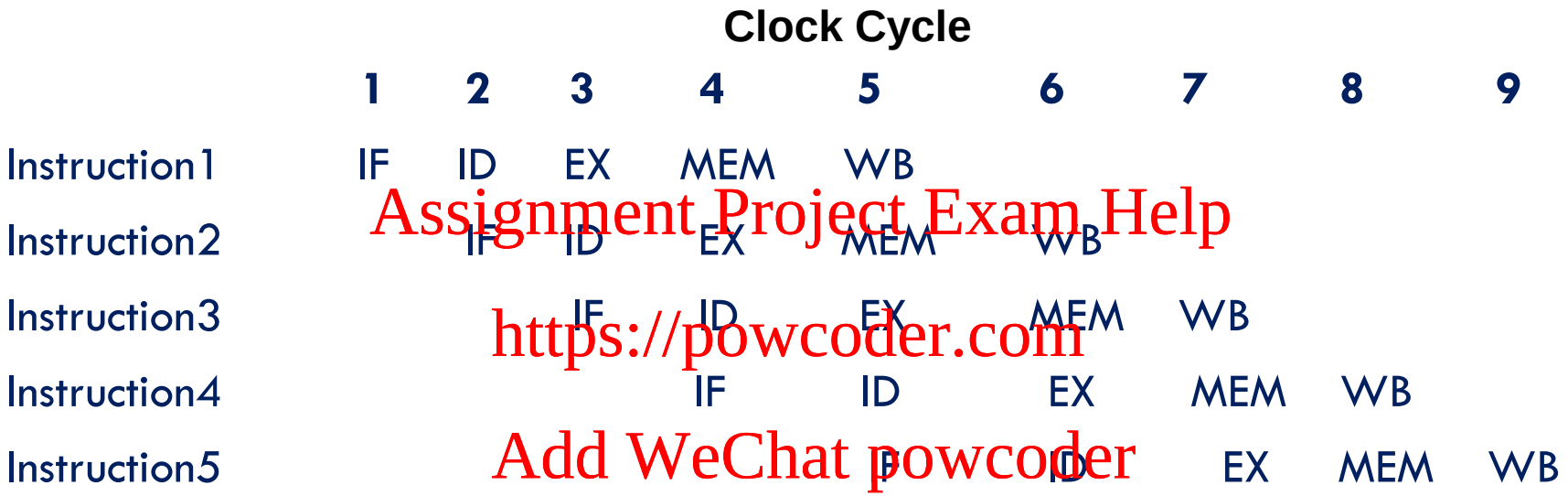
- For instructions that have a result (a destination register), the Write Back writes this result back to the register file

The classic five stage RISC pipeline

- However, not all the instructions need five stages

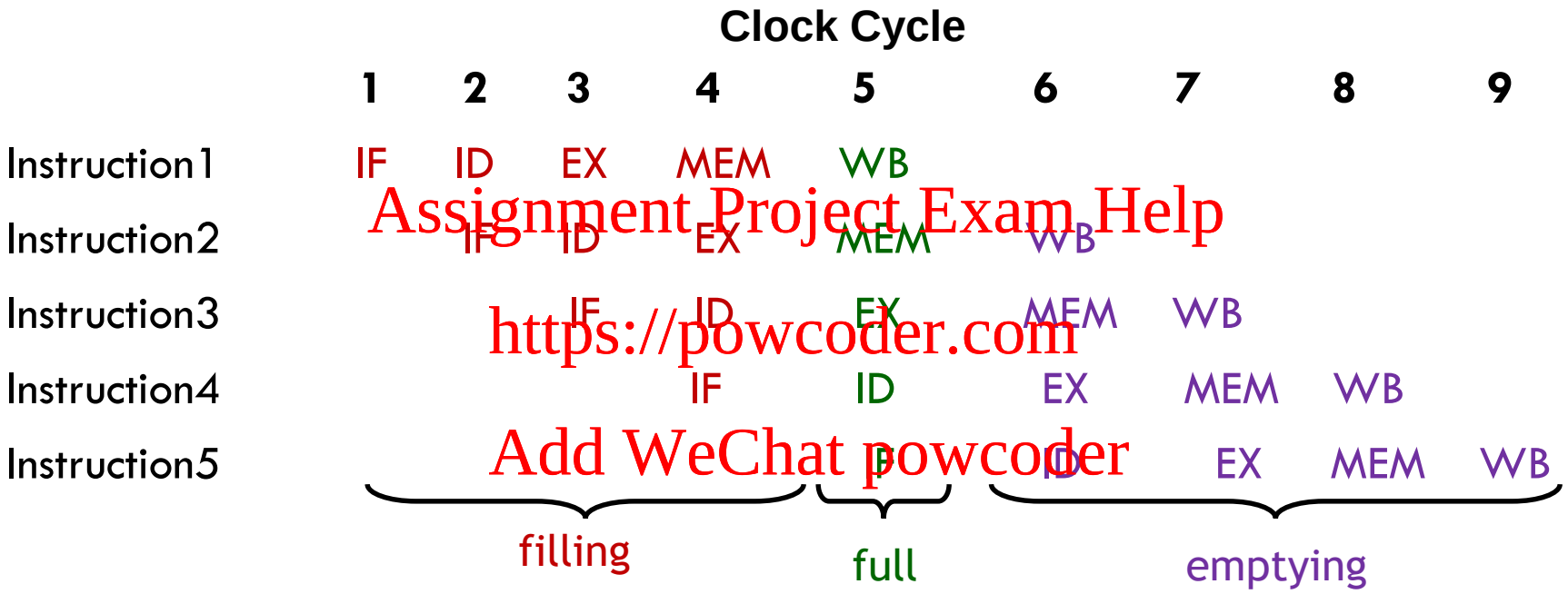
Instruction	Stages Required
Arithmetical	IF ID EX NOP WB
Load	IF ID EX MEM WB
Store	IF ID EX MEM NOP
Branch	IF ID EX NOP NOP

Pipeline Terminology (1)



- 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
- The pipeline depth is the number of stages - in this case, five

Pipeline Terminology (2)



- In the first four cycles here, the pipeline is filling, since there are unused functional units
- In cycle 5, the pipeline is full. Five instructions are being executed simultaneously, so all hardware units are in use.
- In cycles 6-9, the pipeline is emptying

Instruction Pipeline - Wrap Up

- **Pipelining attempts to maximize instruction throughput by overlapping the execution of multiple instructions**

- Pipelining offers significant speedup
  - In the best case, one instruction finishes on every cycle, and the speedup is equal to the pipeline depth

- The pipeline datapath is much like the single-cycle one, but with added pipeline registers
  - Each stage needs its own functional units

Instruction Pipeline - **Hazards**

Limits to pipelining: Hazards prevent next instruction from executing during its designated clock cycle

1. **Structural hazards:** HW cannot support the usage of a function unit to 2 instructions at the same time
2. **Data hazards:** Instruction depends on result of prior instruction still in the pipeline
3. **Control hazards:** Pipelining branch and jump instructions introduce the problem that the destination of the branch is unknown

– Common solution is to **stall** the pipeline until the hazard is resolved, inserting one or more NOP cycles in the pipeline

Instruction Pipeline – Data Hazards (1)

- ↑ **number of data dependences -> ↑ Number of stalls**
- TARGET: Reduce Pipeline Stalls as far as possible

- Typical 5 Pipeline Stages of an integer ALU

- Floating Point ALU, consists of more pipeline stages (more EX stages)
- When an instruction needs the result of another instruction, data hazards occur (RAW, WAR, WAW) => pipeline stalls

Instruction Pipeline – Data Hazards (2)

- Example: add R1,R2,R3
  sub R5,R3,R1

Let us try to pipeline these two instructions.

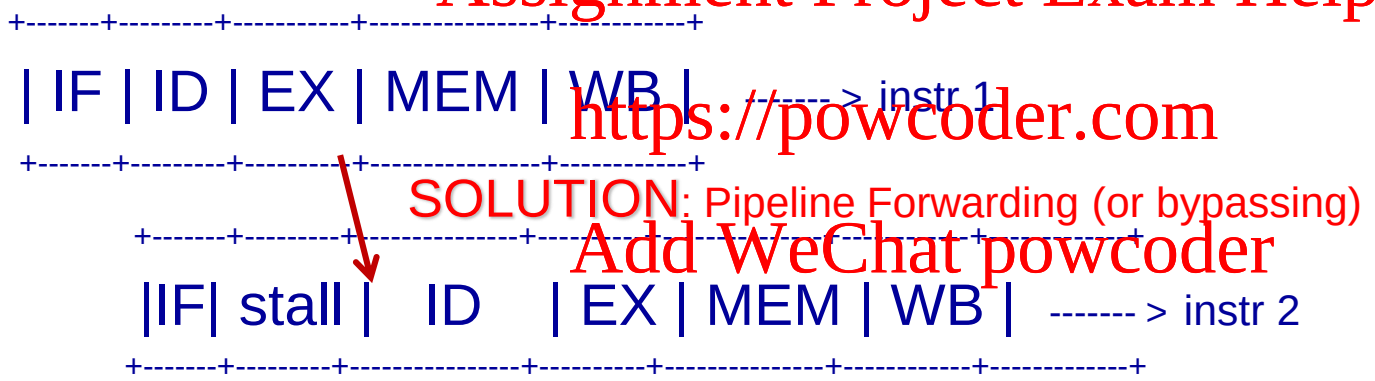
Sub instruction, needs the value of register R1 that will be available after the WB stage.

..and three stalls occur

- SOLUTION: Pipeline Forwarding (or bypassing)

Instruction Pipeline – Data Hazards (3)

- Example: add R1,R2,R3
  sub R5,R3,R1

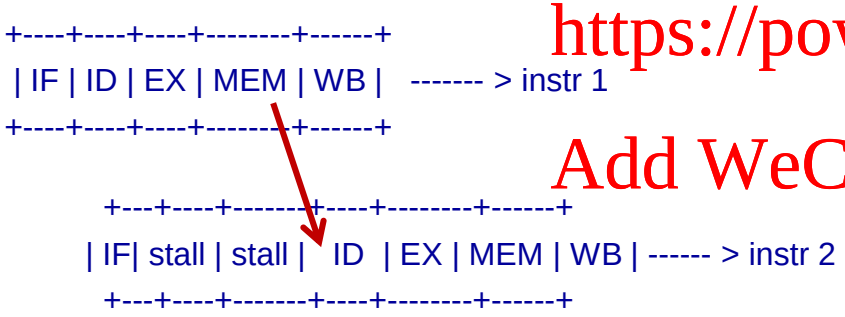


The results of R1 is ready after the EX stage, so it is forwarded

Instruction Pipeline – Data Hazards (4)

- Let us now, consider another example, where bypassing is applied

- ld R1, 0x12FF
- add R3,R4,R1



- Calculation of the address is taking place in the EX stage
- At the next stage, datum read from memory and it is bypassed to the next instruction
- The first stall cycle is inevitable
- *You can find more in https://www.youtube.com/watch?v=EW9vtuthFJY*

Think Pair Share
How many stall cycles occur to the following codes


```
I1:     load  r1,a
I2:     load r2,b
I3:     r3=r1+r2
I4:     load r4,c
I5:     r5=r3-r4
I6:     r6=r3*r5
I7:     st d,r6
```

- In this code pipeline stalls occurs twice. The first one due to immediate read of r2 (i3) after it is loaded form memory (i2), and similarly between i5 and i4.
- In total, 4 CPU cycles are wasted

Solution: Reorder instructions as follows

```
I1:     load  r1,a
I2:     load r2,b
I4:     load r4,c  -> hiding the latency from i2 to i3..
I3:     r3=r1+r2
I5:     r5=r3-r4
I6:     r6=r3*r5
I7:     st d,r6
```

In this code, no stalls occur, since none of the load instructions is immediately followed by a dependent (arithmetic) instruction

Control Hazards

- A control hazard is when we need to find the destination of a branch, and can't fetch any new instructions until we know that destination

beq r1,r3, L1	***IF***	***ID***	***EX***	***MEM***	***WB***
and r2,r3,r5	***NOP***	***NOP***	***NOP***	***IF?***	
or r6,r1,r7					
add r8,r1,r9					
L1: xor r10,r1,r11	***NOP***	***NOP***	***NOP***	***IF?***	

Only at that cycle the CPU knows which is the next instruction.

Solutions:

✓ **Branch Prediction:** The outcome and target of conditional branches are predicted using some heuristic

- Branch predictors play a critical role in modern CPUs

Control Hazards

- **Without branch prediction,** the processor **would have to wait until the conditional jump instruction has passed the execute stage** before the next instruction can enter the fetch stage in the pipeline.
- The branch predictor attempts to **avoid this waste of time by trying to guess** whether the conditional jump is most likely to be taken or not taken
- The branch that is guessed to be the most likely is then fetched and speculatively executed
- If it is later detected that the guess was wrong then the speculatively executed or partially executed instructions are discarded and the pipeline starts over with the correct branch, incurring a delay

Avoid writing code with if conditions