CE1006/CZ1006
Computer Organisation and Architecture

Introduction to Parallelism

Oh Hong Lye
Lecturer
SCSE, Nanyang Technological University.

Introduction

- **Parallelism** means doing multiple things at the same time – **more work** can be done at the same time, but at the cost of **more resources**

Less Work Done

How can we realise parallelism in computers and what are the issues?

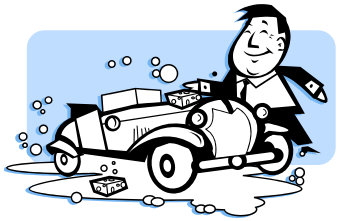
More Work Done in the same amount of time, but more resources required

Non-Pipeline Example - Car Wash

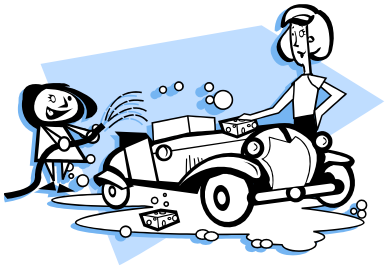
Source: http://www.warthman.com/visual-analogies.htm

Notice that in each time unit, only **one** sub-task is active (other 4 sub-tasks are idle)

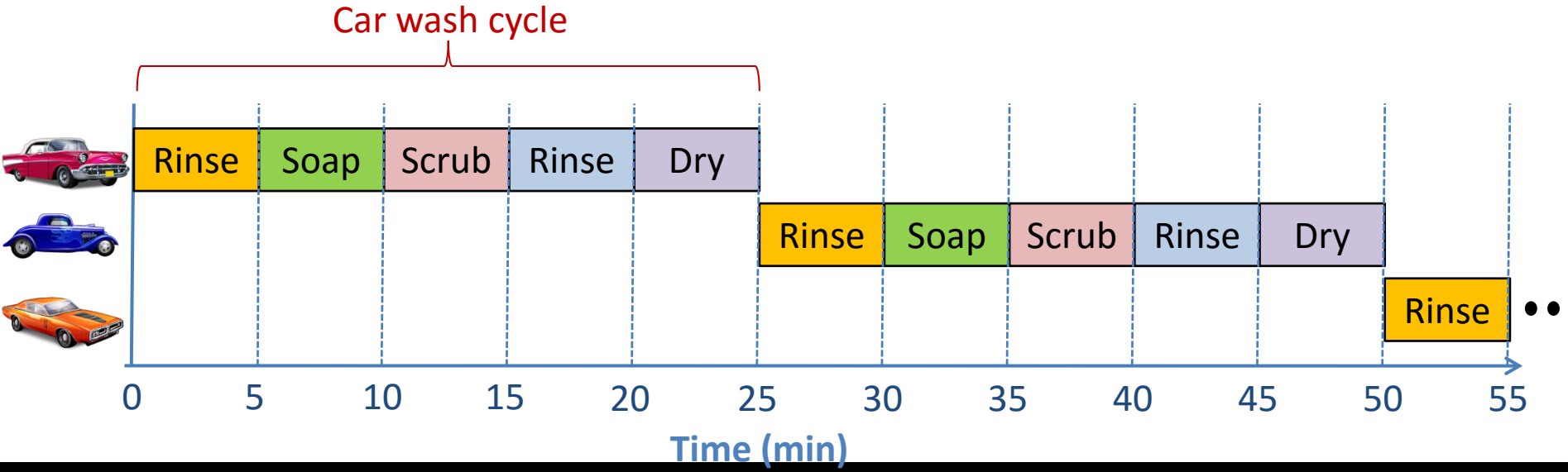

Rinse


Soap


Scrub

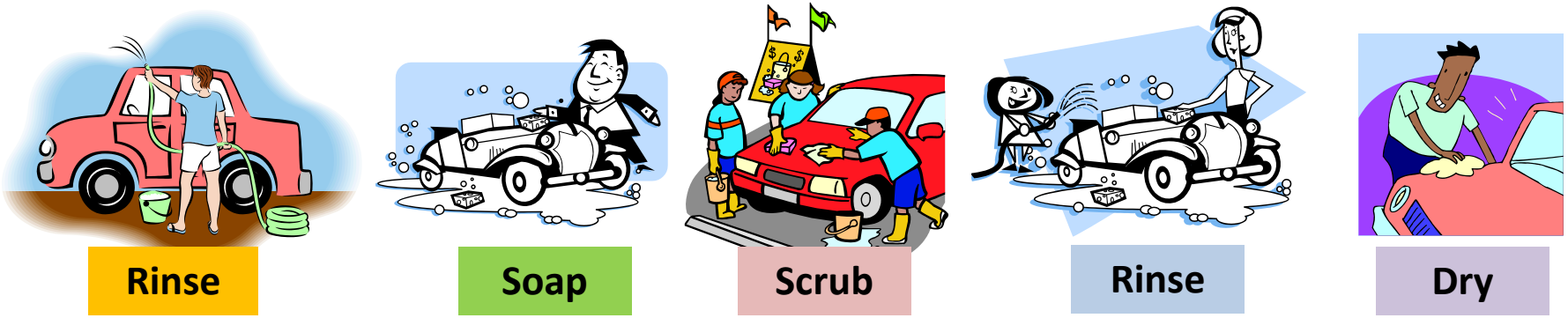

Rinse

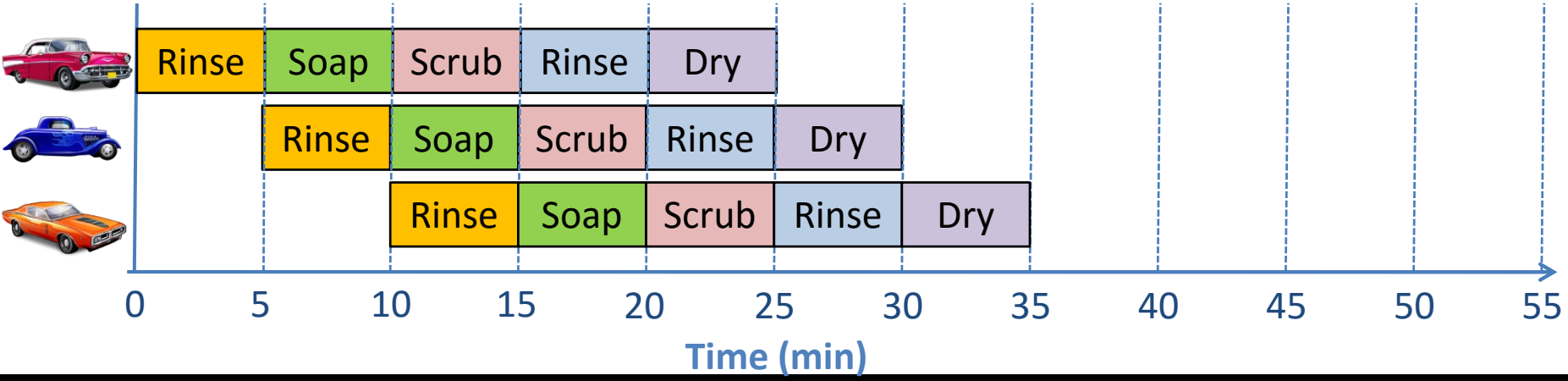

Dry



Pipeline Example - Car Wash

- **Pipelining** allows **multiple** sub-tasks to be carried out **simultaneously** using **independent resources**
  - Need to **balance** time taken by each sub-task



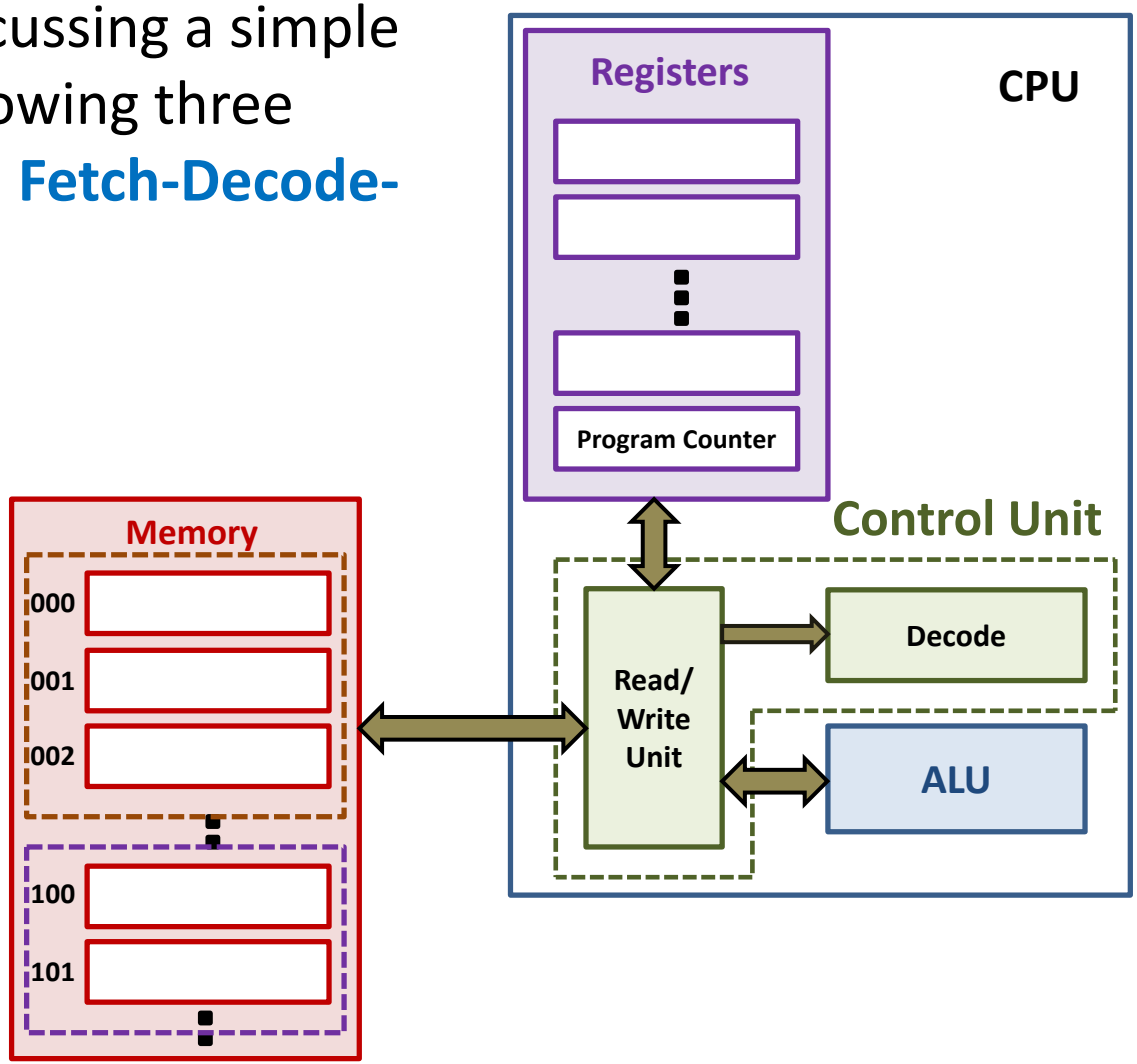


Review of a Simple CPU

- So far we have been discussing a simple CPU which does the following three operations **sequentially**: **Fetch-Decode-Execute**

  - In the simple CPU, the **Fetch-Decode-Execute** cycle of an instruction must complete before the next instruction is fetched

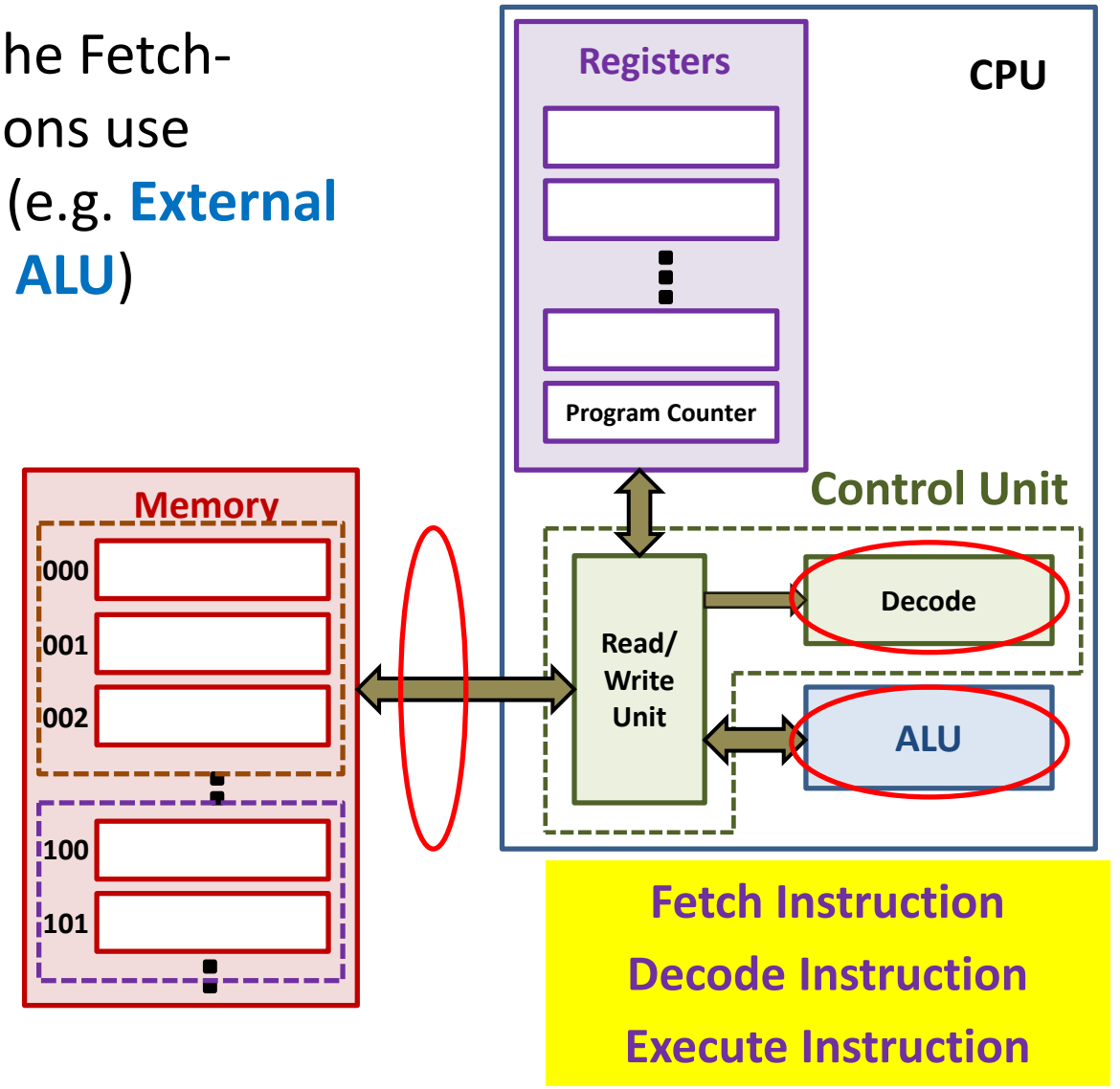
  - Is it possible to use **pipelining** to **improve performance?**



Pipelining

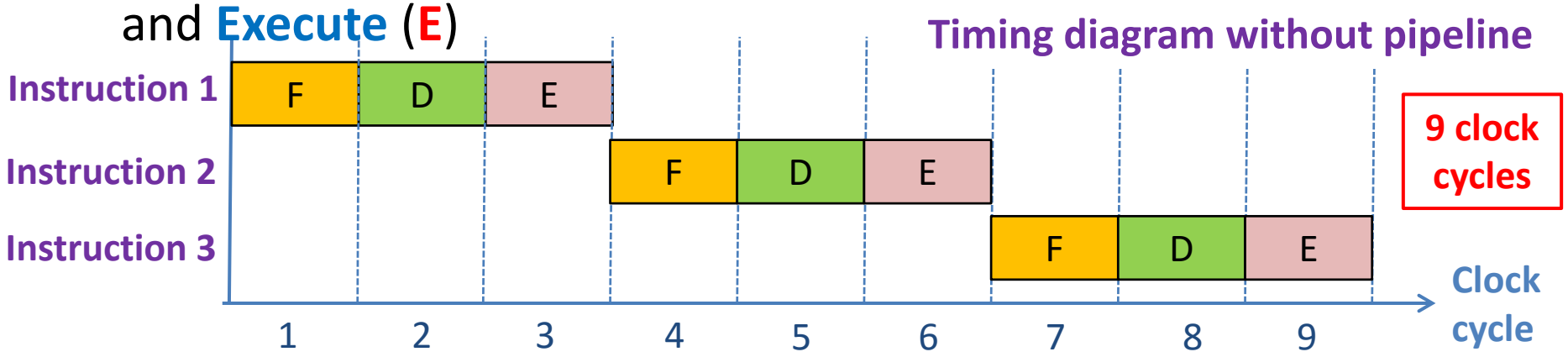
- Pipelining is possible if the Fetch-Decode-Execute operations use **independent resources** (e.g. **External buses**, **Control Unit** and **ALU**)

- With some modifications to the CPU hardware, it is possible to overlap the **Fetch-Decode-Execute** cycles of different instructions

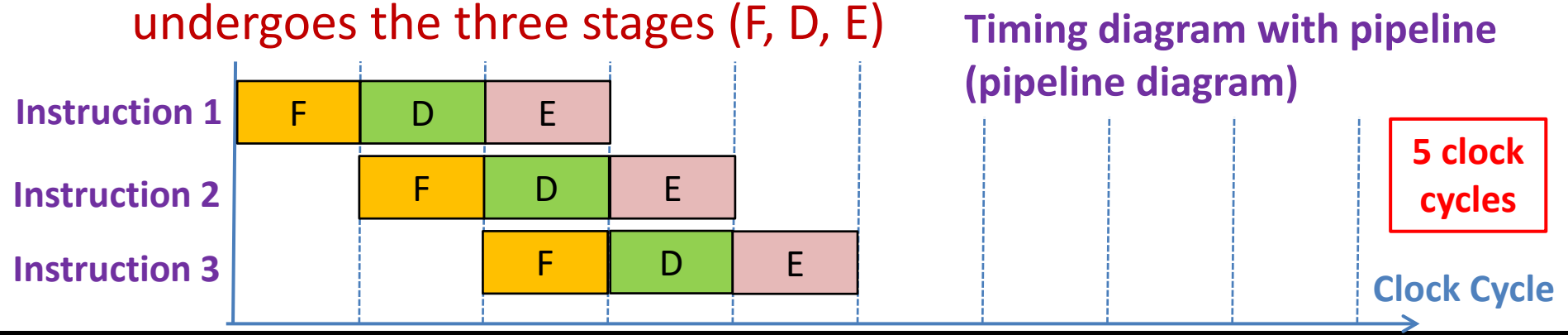


Pipelining Example

- Consider a processor without pipeline. Each instruction is broken down into the following stages: **Fetch Instruction** (F), **Decode** (D), and **Execute** (E)



- Now, consider a processor organisation with **3** pipeline stages
  - Assume each stage takes a unit of time and all instructions undergoes the three stages (F, D, E)



CE1006/CZ1006
Computer Organisation and Architecture

Pipeline Resource and Data Conflicts

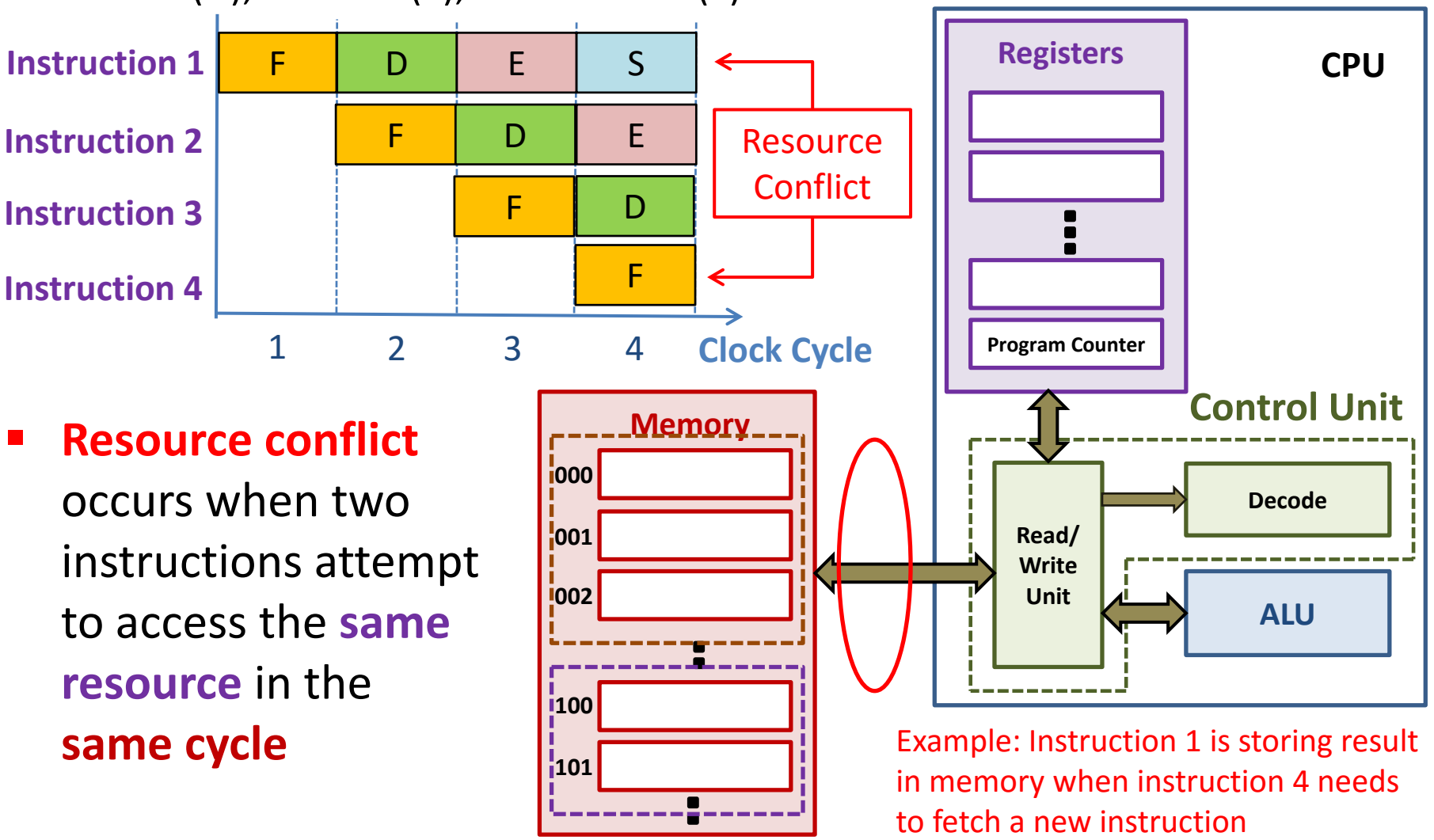
Oh Hong Lye
Lecturer
SCSE, Nanyang Technological University.

Pipelining Conflicts (Hazards)

- The 3 stage pipelining example on the previous slide shows the **Minimum Time** for the three instructions to complete
- In order to achieve **minimum time**, we assumed that there are no **Pipeline Conflicts**
- However, **pipeline conflicts** due to the following conditions can affect the performance of the pipelined processor:
  - Resource conflicts
  - Data dependencies
  - Branch statements
- Measures can be taken at the **software** level as well as at the **hardware** level to **reduce the effects of these conflicts**, but they cannot be totally eliminated

Resource Conflicts

- Let's now consider a processor with 4 pipeline stages: **Fetch Instruction** (**F**), **Decode** (**D**), **Execute** (**E**), **Store Result** (**S**)

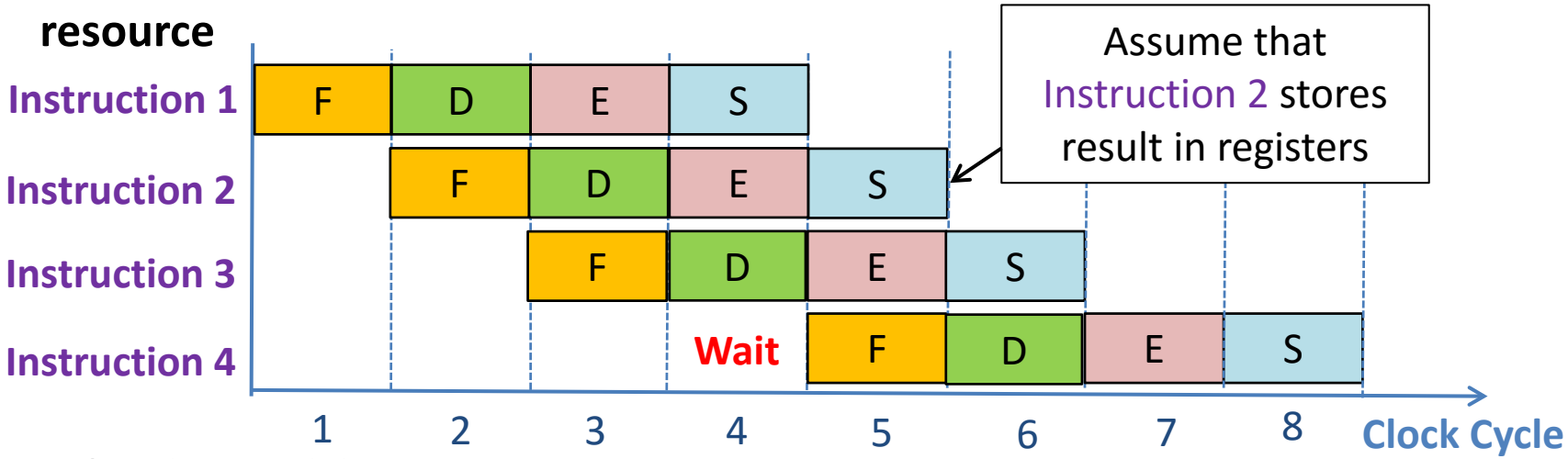


- **Resource conflict** occurs when two instructions attempt to access the **same resource** in the **same cycle**

Example: Instruction 1 is storing result in memory when instruction 4 needs to fetch a new instruction

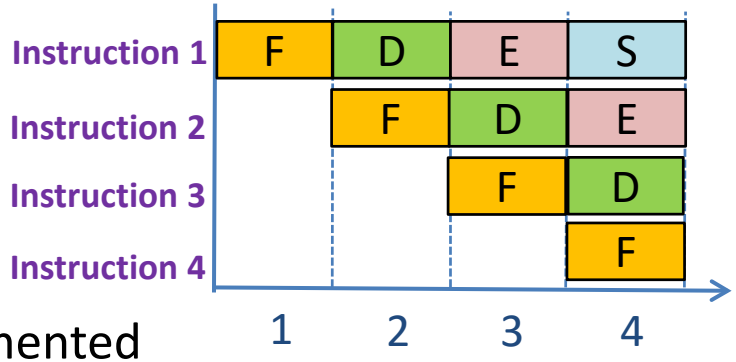
Resolving Resource Conflicts

- **Method 1: Wait**
  - **Require mechanism to detect the conflict and delay instruction access to resource**



- **Method 2: Add resources**
  - **Add more hardware to eliminate resource conflict**
  - **In this example, we can use two memories with two memory ports**
    - **Instruction memory**
    - **Data memory**

} Can be implemented as caches



Data Dependencies

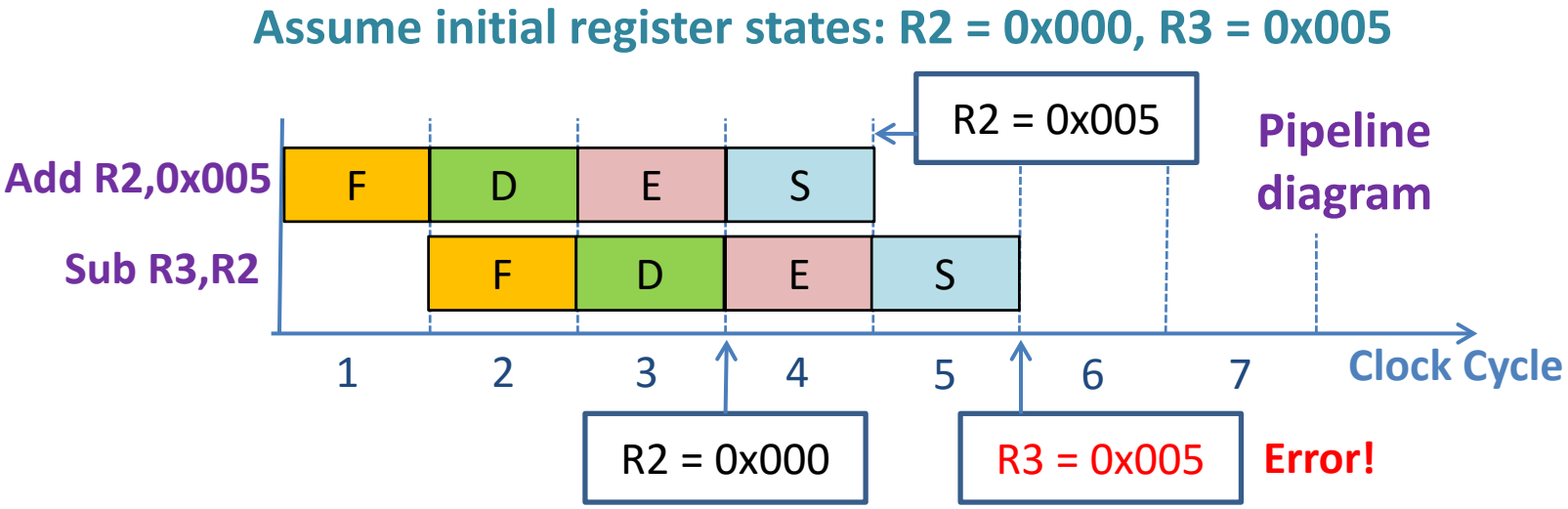
- Given two instructions **I1** and **I2**, where instruction **I2** is executed after instruction **I1**

```
Add R2,#0x005  ;R2 = R2+0x005 (I1)
Sub R3,R2      ;R3 = R3-R2    (I2)
```

- **Data dependency** arises because the destination register of **I1**, which is **R2**, is also the source operand of **I2**
- In a **non-pipelined processor**, result is available in **R2** because **I1** ***always*** completes before **I2**
- **Data conflict** occurs in a **4 stage pipeline processor** as **I2** is executed immediately after **I1**, and the required result in **R2** is not yet available to be used in the **Execute** stage of **I2**

Data Conflict

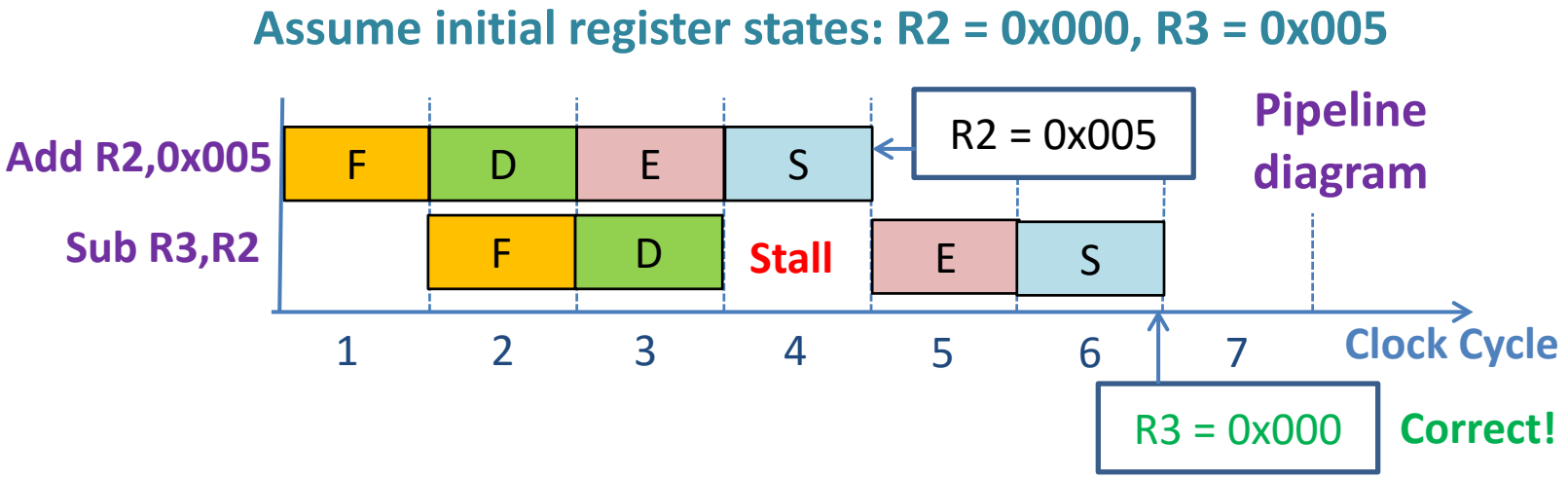
- In a **4-stage pipeline**, the old value is still in **R2** when ***Subtract*** is in **Execute** (E) stage



- Two possible methods for resolving data conflict:
  - Stall the pipeline
  - Insert NOP instructions

Resolving Data Conflict – Stall the Pipeline

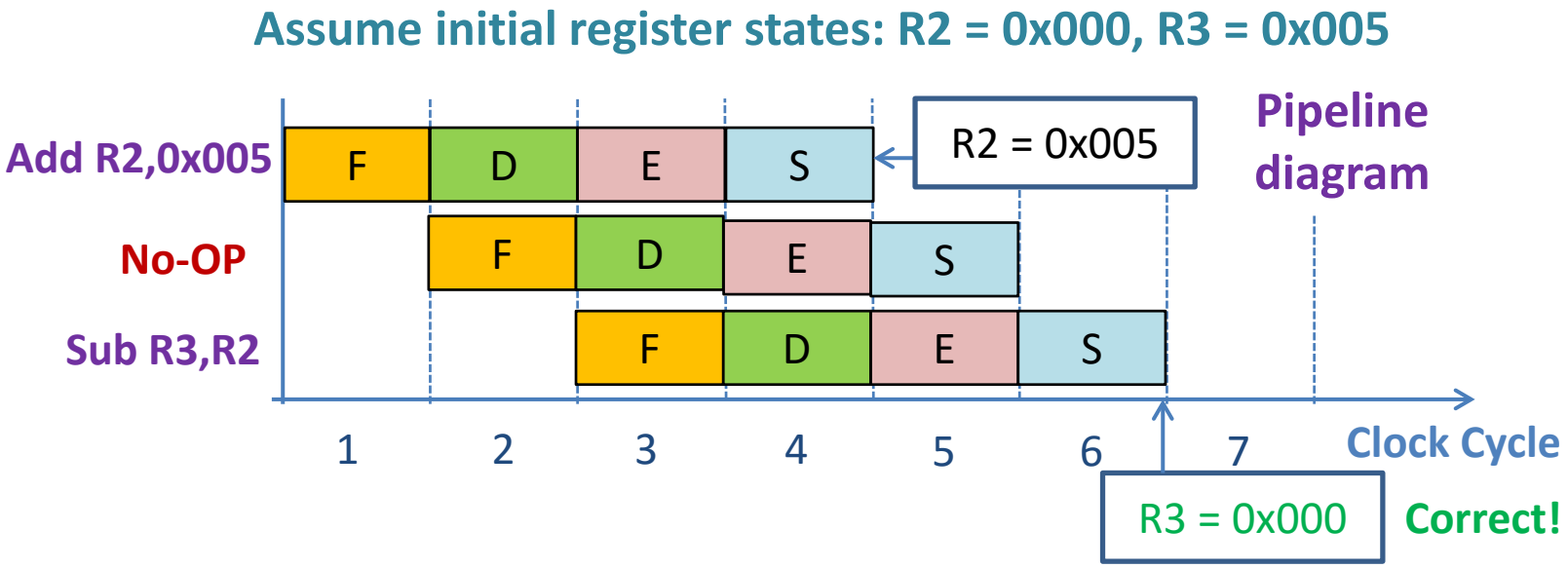
- **Hardware circuitry** can be used to detect data dependency between instructions
  - Compare destination identifier in the Execute Stage with source(s) in the Decode stage



- If data dependency is detected (i.e. **R2** matches), allow ***Add*** to continue normally, but stall the **Decode** stage of ***Subtract***
- After ***Add*** completes, ***Subtract*** is allowed to resume

Resolving Data Conflict – Insert NOP Instructions

- **Compiler can analyze and insert redundant instructions to reduce data conflict**
  - **Data dependencies are evident in instructions during compilation**
  - **Compiler inserts explicit NOP (No Operation) instructions between instructions with data dependencies**
- **Delay ensures new value is available in register but causes total execution time to increase**



CE1006/CZ1006
Computer Organisation and Architecture

Pipeline Branch Operation

Oh Hong Lye
Lecturer
SCSE, Nanyang Technological University.

Branch Statements

- Usually need to perform two jobs:
  - Evaluate condition to determine if **branch** should be **taken/not taken**
  - If branch is taken, calculate **branch target** using adder in **ALU**

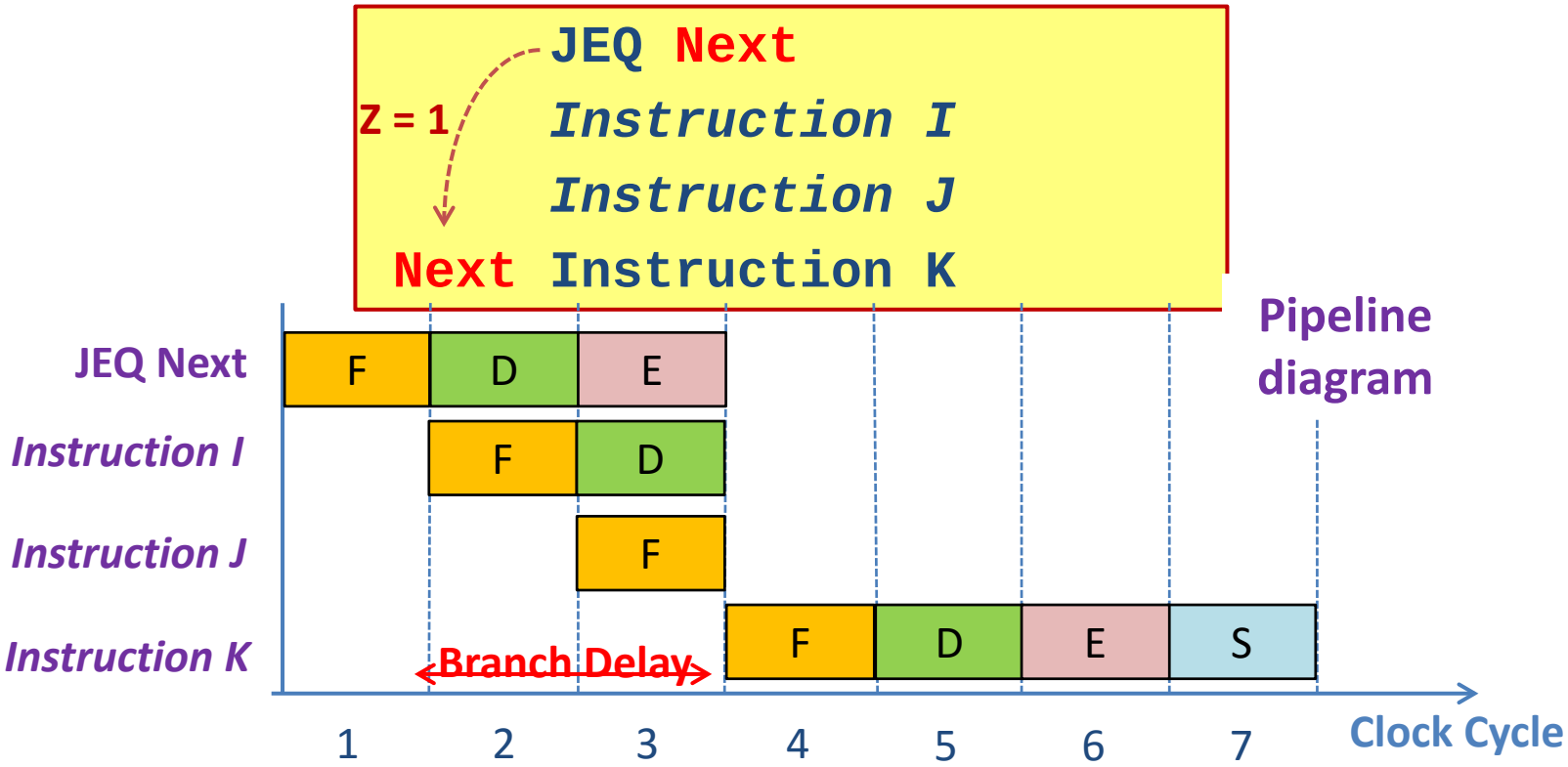
Bits	11 to 8	7 to 4	3 to 0	
Hex	mnemonic	destination	source	operation
B	JMP = BRA	-128 to +127		PC ← PC ± n
C	JEQ = JZ	-128 to +127		If Z=1, PC ← PC ± n
D	JNE = JNZ	-128 to +127		If Z=0, PC ← PC ± n
E	JHS = JC	-128 to +127		If C=1, PC ← PC ± n
F	JLO = JNC	-128 to +127		If C=0, PC ← PC ± n

Bits	x: 7 to 4	y: 3 to 0	
Hex	mnemonic	s/d/n	operation
8	JDAR	n, -8 to +7	AR ← AR-1, if AR != 0, PC ← PC ± n
9	JPE	n, -8 to +7	If parity of AR is even, PC ← PC ± n
D	JLT	n, -8 to +7	If N != V, PC ← PC ± n
E	JGT	n, -8 to +7	If Z = 0 and N = V, PC ← PC ± n

- These two jobs are usually done in the **Execute** stage
- However in pipelining, by the time the branch target is calculated, unnecessary instructions would have been fetched and decoded – resulting in **Branch Delay**

Branch Delay

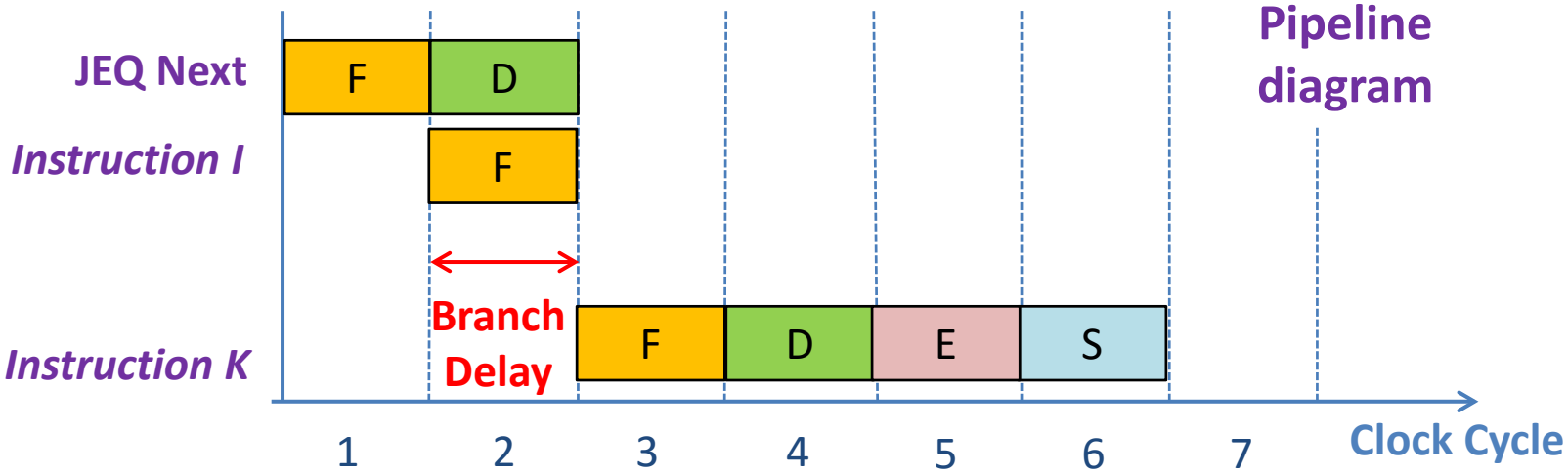
- **Branch statements in pipelining can lead to Branch Delay which cause significant performance loss**



- **The branch target is only known after the Execute stage, but by this time, Instructions I and *J* have already been fetched**
  - **Instructions I and J will be discarded, resulting in two-cycle branch delay**

Reducing Branch Delay

- Branch delay can be reduced by determining the branch decision (taken or not taken) and calculating the branch target to the **Decode** stage
  - **An additional adder is introduce in the Decode stage to enable earlier calculation of branch target**



- After the Decode stage, the branch decision and the branch target is known
  - **Hence if branch is taken, only *Instruction I* needs to be discarded**
  - **Branch delay is reduced to one cycle**

Delayed Branching

- In the previous implementation, the **instruction immediately following a branch is always fetched**, regardless of the branch decision
- If **branch is taken**, this next instruction will be discarded resulting in **branch delay**
- **Delayed Branching** is a method that ensures **no** instructions are discarded after the branch
- **Compiler** can schedule instructions such that an **independent instruction** is inserted after the branch
  - **If such independent instruction exists, they will always be executed, leading to zero branch delay**
  - **If an independent instruction cannot be found, NOP instructions are inserted after the branch resulting in one-cycle branch delay**

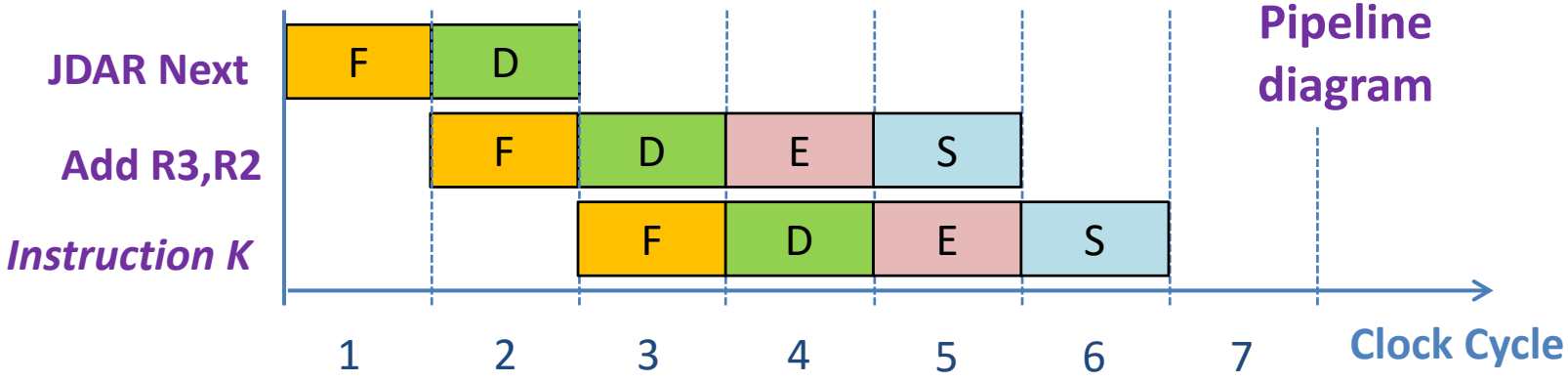
Delayed Branching
- Scheduling the Branch Delay Slot

- Compiler finds and moves an **independent instruction** before the branch into the slot after the branch

```
        Add R3,R2
        JDAR Next
        Instruction I
        . . .
Next    Instruction K
```

```
        JDAR Next
        Add R3,R2
        Instruction I
        . . .
Next    Instruction K
```

- Instead of conditionally discarding instruction, **always** let it **complete execution**



Dynamic Branch Prediction

- Hardware circuitry to guess outcome of a conditional branch
  - **Branch history table is implemented to store target addresses of taken branches**
- If prediction is **correct**
  - **Continue normal execution – no wasted cycles**
- If prediction is **incorrect**
  - **Flush instructions that were incorrectly fetched – wasted cycles**
  - **Update prediction bit and target address for future use**

Branch penalty is more significant in deeper pipelines!