Instruction Pipeline: Introduction

Introduction

- The earliest use of parallelism in designing CPUs (since 1985) to enhance processing speed was Pipelining
- Computers execute billions of instructions, so instruction throughput is what matters
- It is an implementation technique whereby multiple instructions are executed in an overlapped manner
- It takes advantage of temporal parallelism that exists among the actions needed to execute an instruction

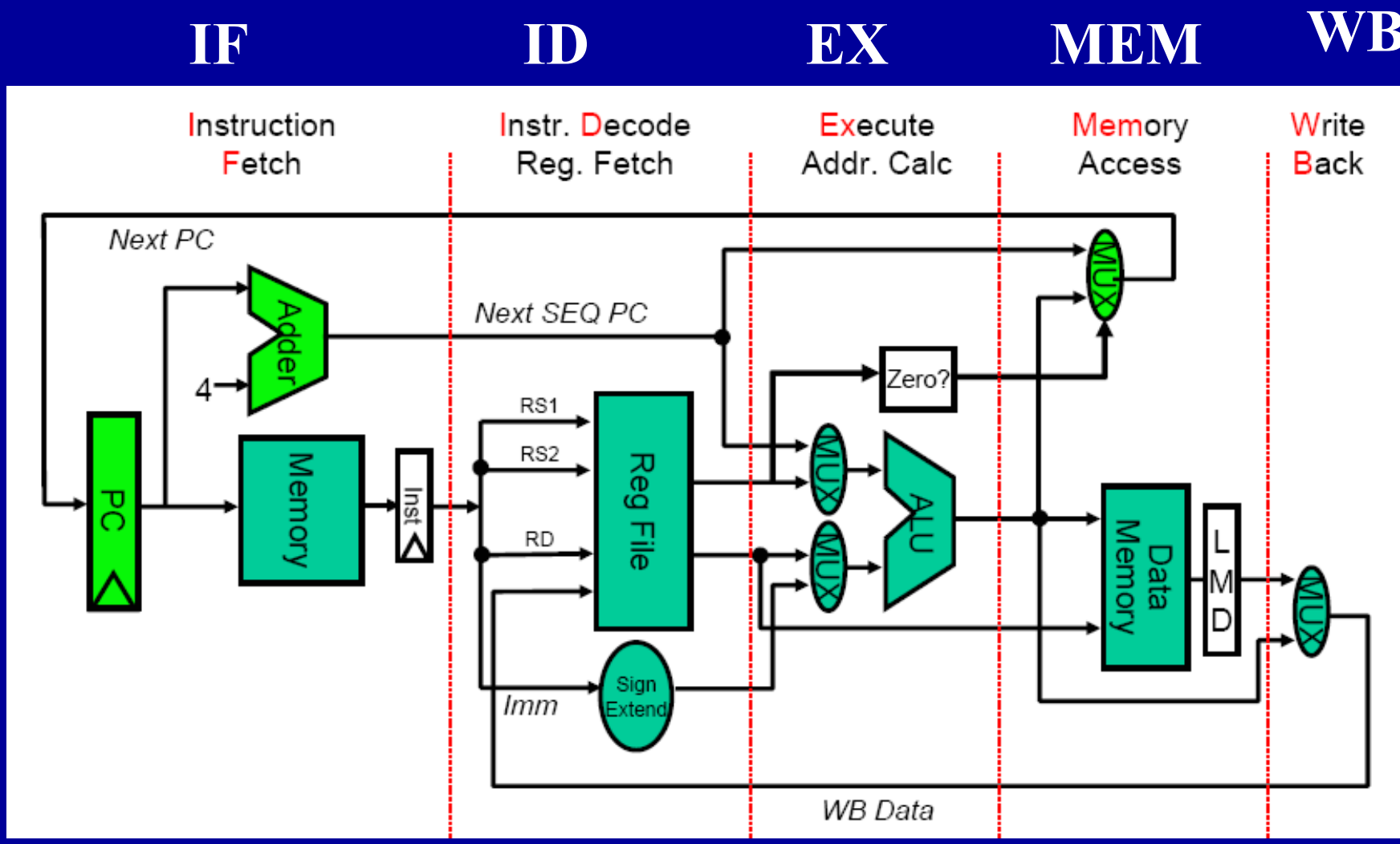
Ideal Conditions

- **Assumptions:**
- Instructions can be divided into independent parts, each taking nearly equal time
- Instructions are executed in sequence one after the other in the order in which they are written
- Successive instructions are independent of one another
- There is no resources constraint

Implementing Instruction Pipeline

- Divide instruction execution across several stages
- Each stage accesses only a subset of the CPU's resources
- Different instructions are in different stages simultaneously
- Ideally, a new instruction can be issued in every cycle
- Cycle time is determined by the longest stage

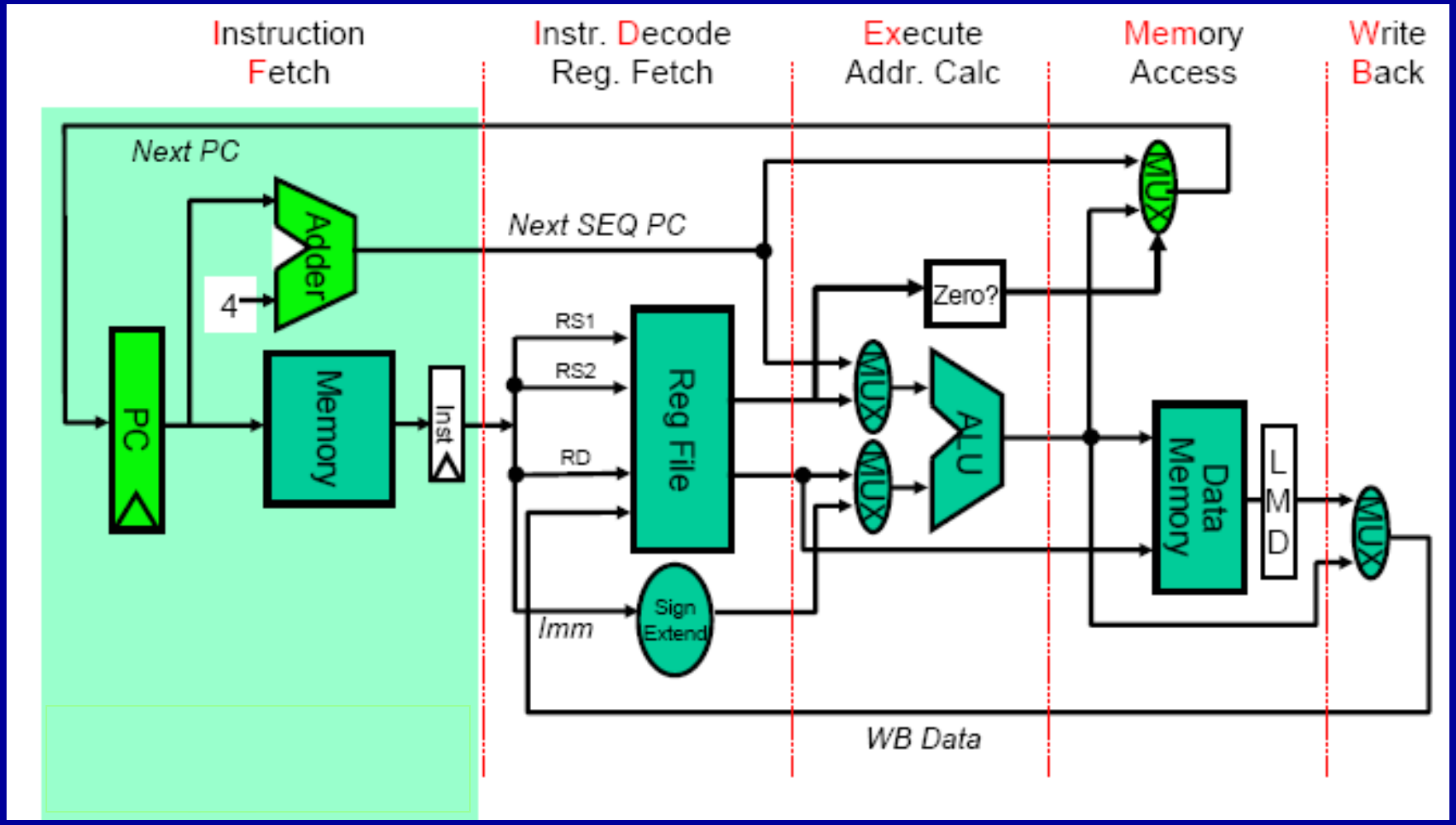
Simple RISC Datapath



Pipelining of a RISC-like Processor

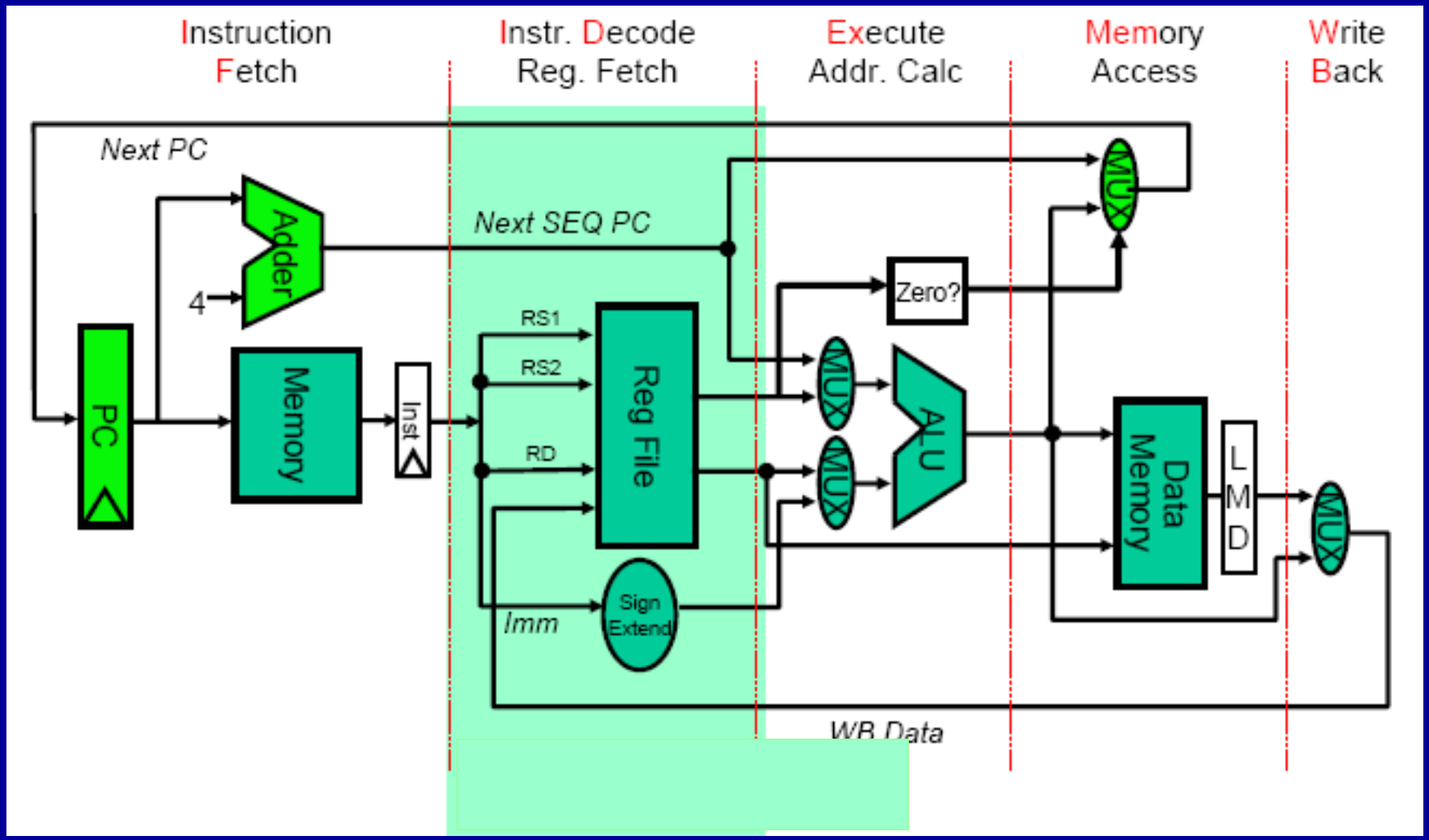
- All ALU operations are performed on register operands
- Separate Instruction and data memory
- Only instructions which access memory are load and store instructions
- Instructions can be broken into the following five parts:
  - Instruction Fetch from instruction memory
  - Instruction decode and operand read
  - Instruction Execution
  - Load/store operands (data memory)
  - Write back results in the registers

Instruction Fetch



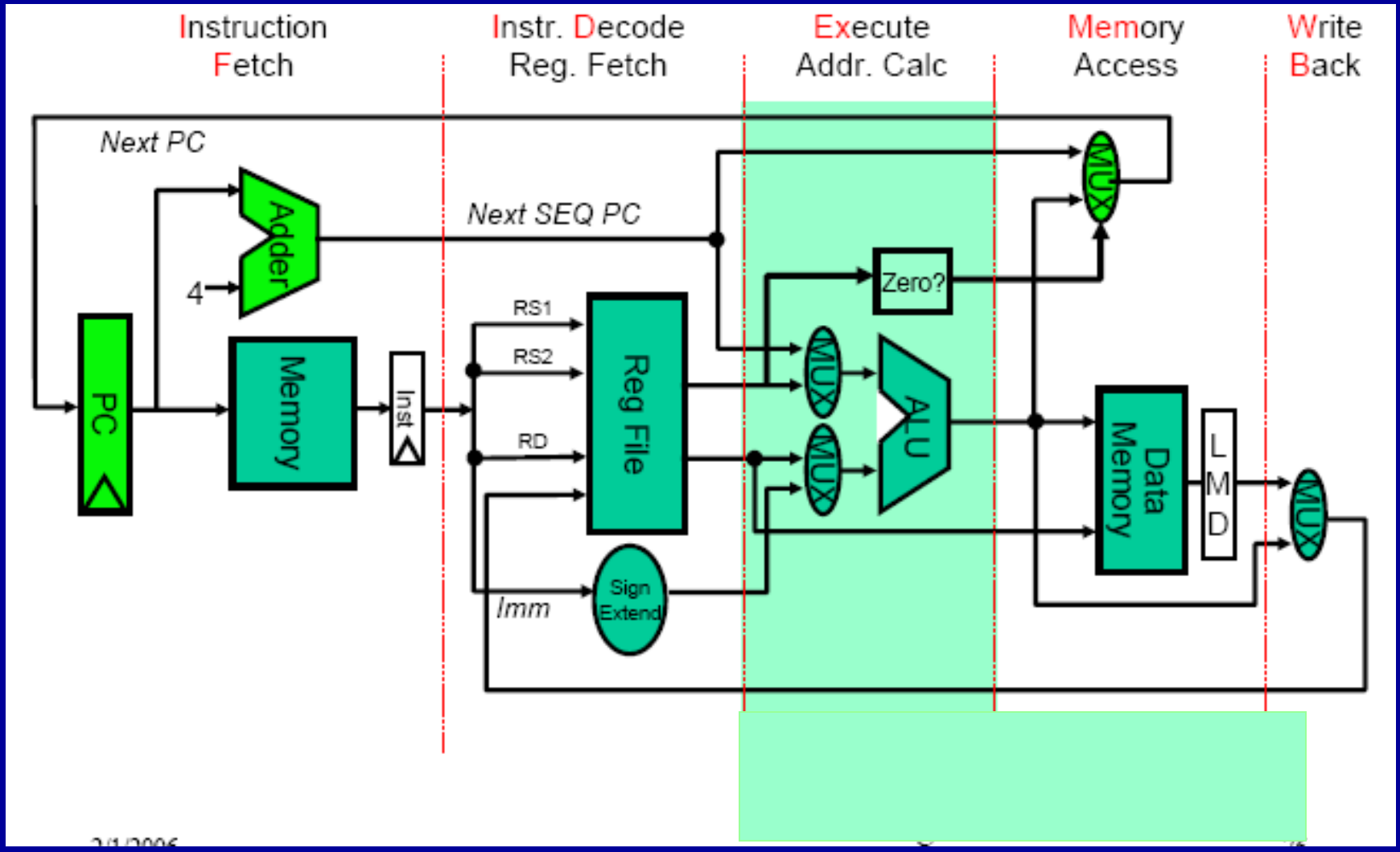
IF Cycle: IR← Mem [PC]; NPC ← PC + 4

Instruction Decode



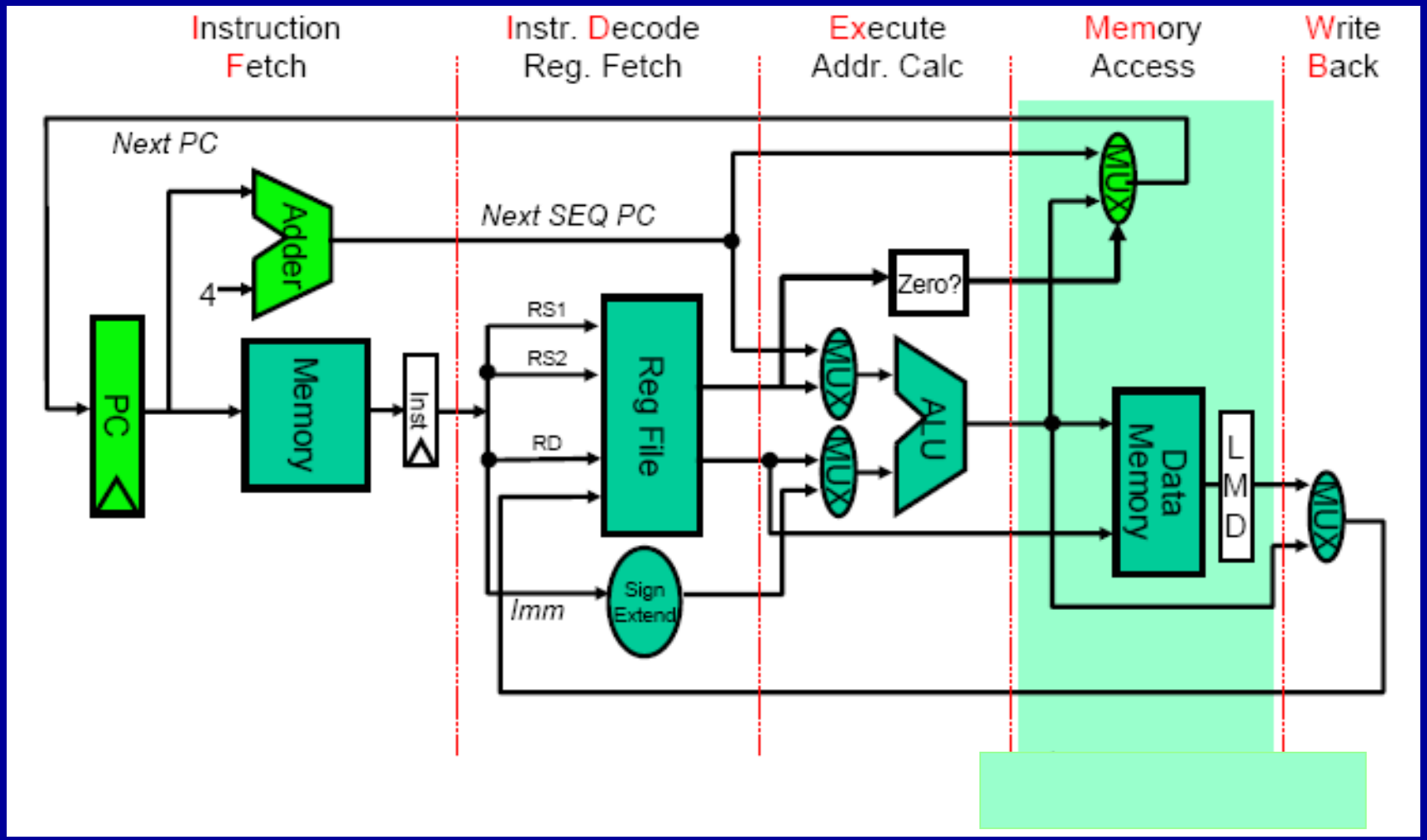
ID Cycle: A ← Regs [IR 6-10]; B← Regs [IR 11-15]; Imm← ((IR16)16 # # IR 16-31)

Execute



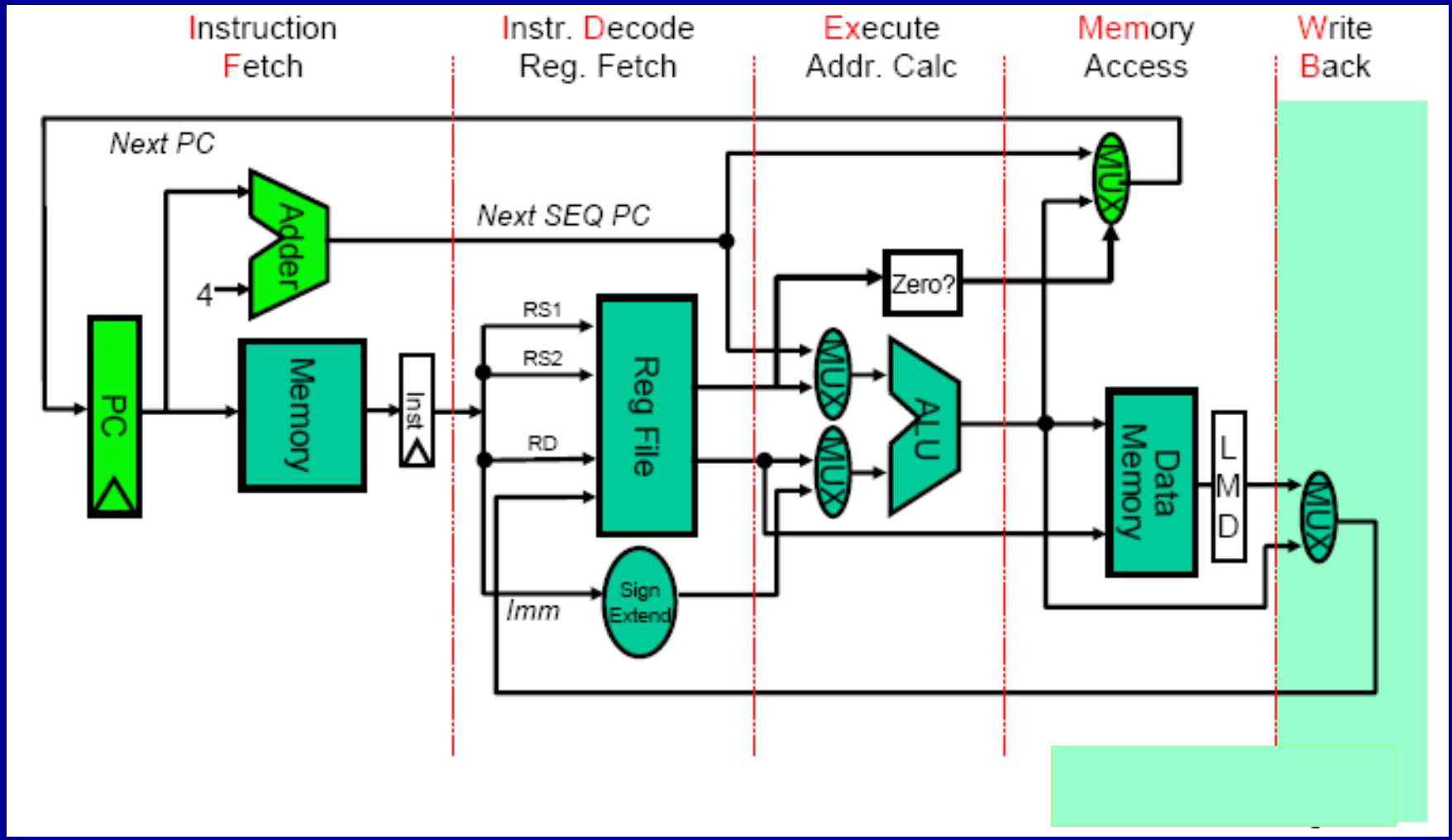
EX Cycle: ALUoutput ← A + Imm
or ALUoutput ← A func B or ALUoutput ← A op Imm
or ALUoutput ← NPC + Imm; Cond ← (A op 0)

Memory access



MEM Cycle: LMD← Mem [ALUoutput]
or Mem [ALUoutput] ← B
or If (Cond) PC ← ALUoutput ; Else PC← NPC

Write Back



WB Cycle: Regs [IR 16-20]←ALUoutput
or Regs [IR 11-15]←ALUoutput
or Regs [IR 11-15]←LMD

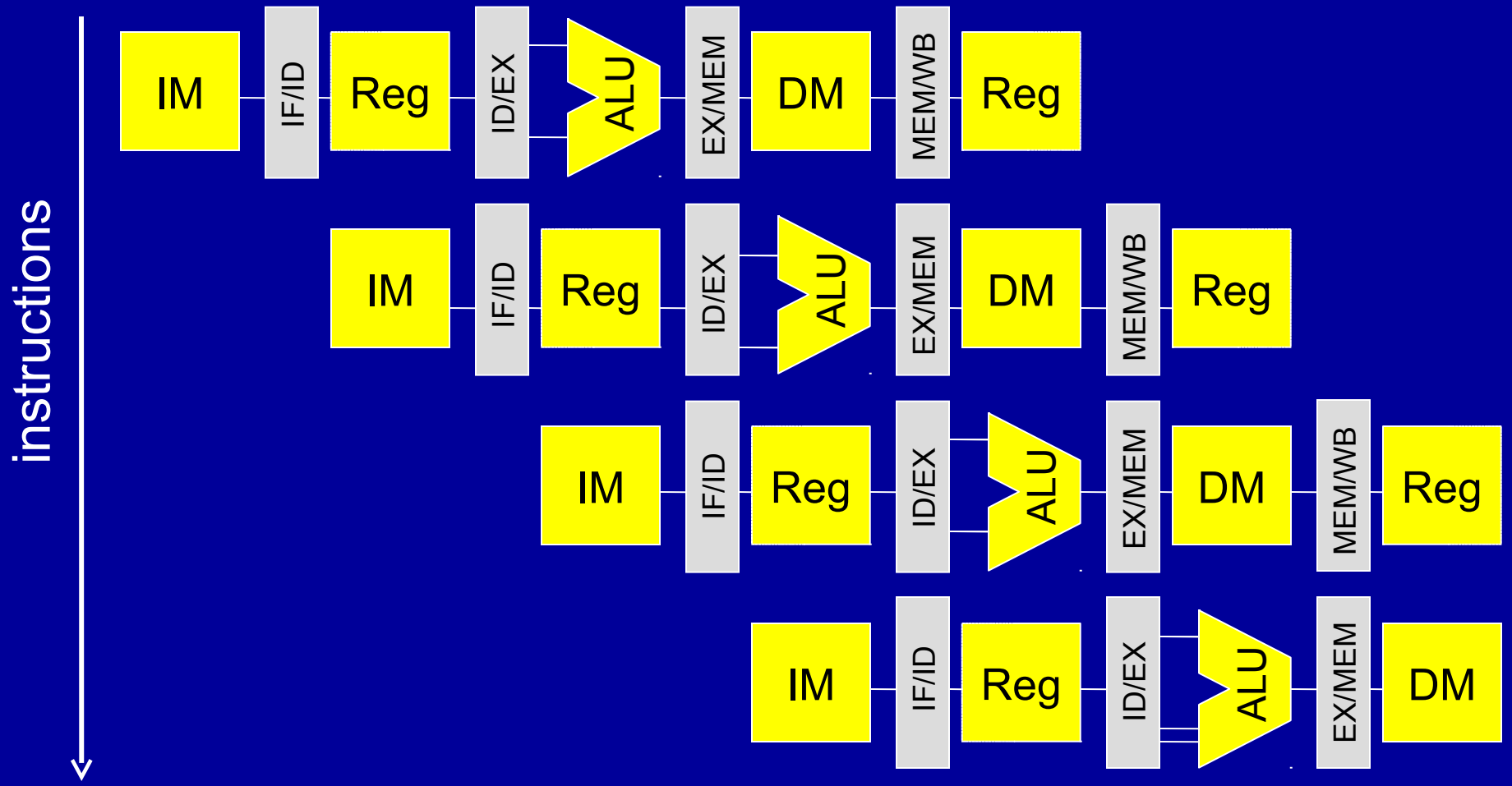
CPI for the Multiple-Cycle Implementation

- Branches and stores: 4 cycles (no WB), all other instructions: 5 cycles
- If 20% of the instructions are branches or loads, CPI = 0.8*5 + 0.2*4 = 4.80
- ALU operations can be allowed to complete in 4 cycles (no MEM)
- If 40% of the instructions are ALU operations, CPI = 0.4*5 + 0.6*4 = 4.40
- Pipelining the implementation can help to reduce the CPI

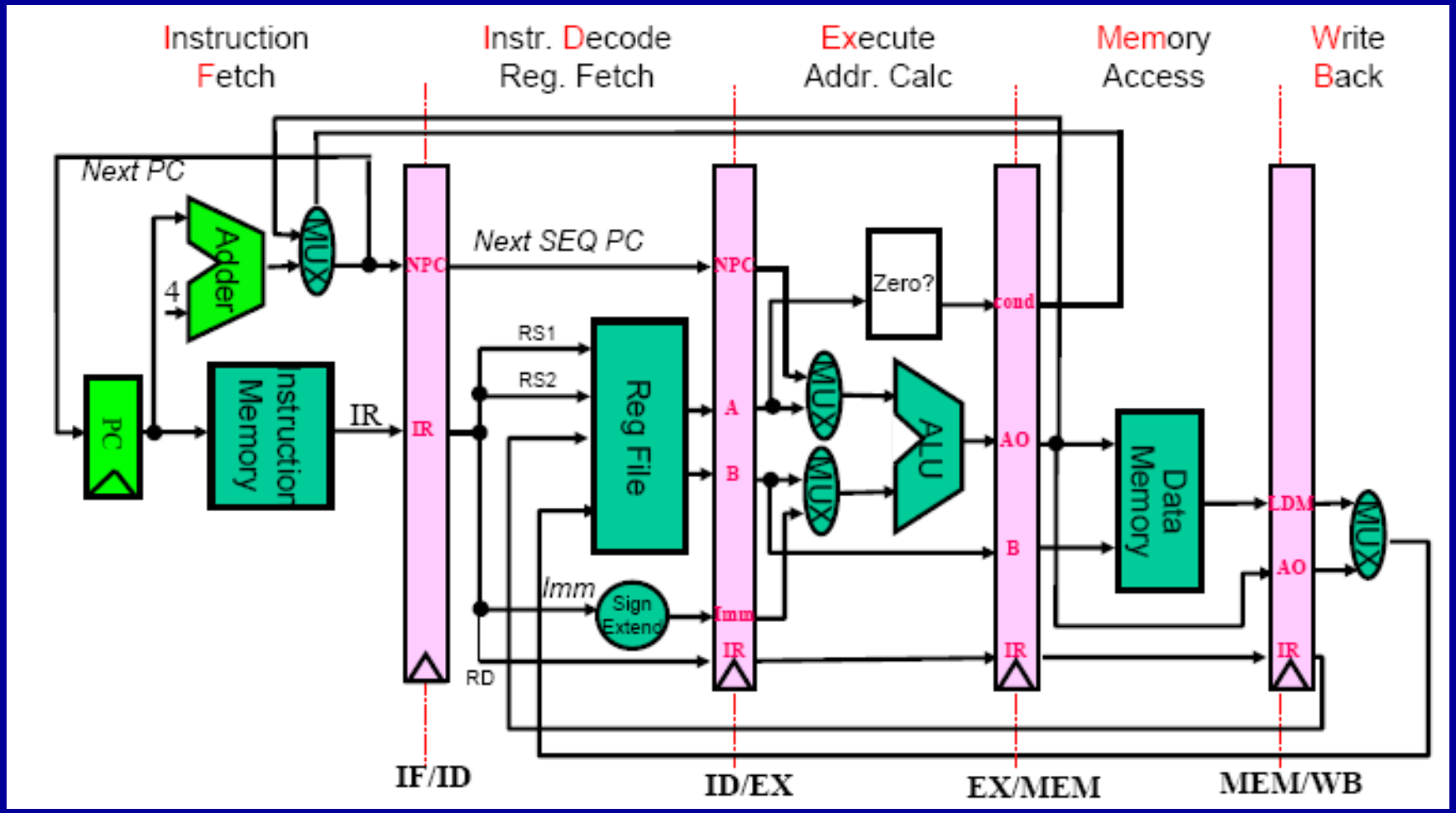
Pipeline Registers

- **Pipeline registers** are essential part of pipelines:
  - There are 4 groups of pipeline registers in the 5 stage pipeline.
- Each group saves output from one stage and passes it as input to the next stage:
  - IF/ID
  - ID/EX
  - EX/MEM
  - MEM/WB
- This way, each time “something is computed”...
  - Effective address, Immediate value, Register content, etc.
  - It is saved safely in the context of the instruction that needs it.

Pipeline Register Depiction

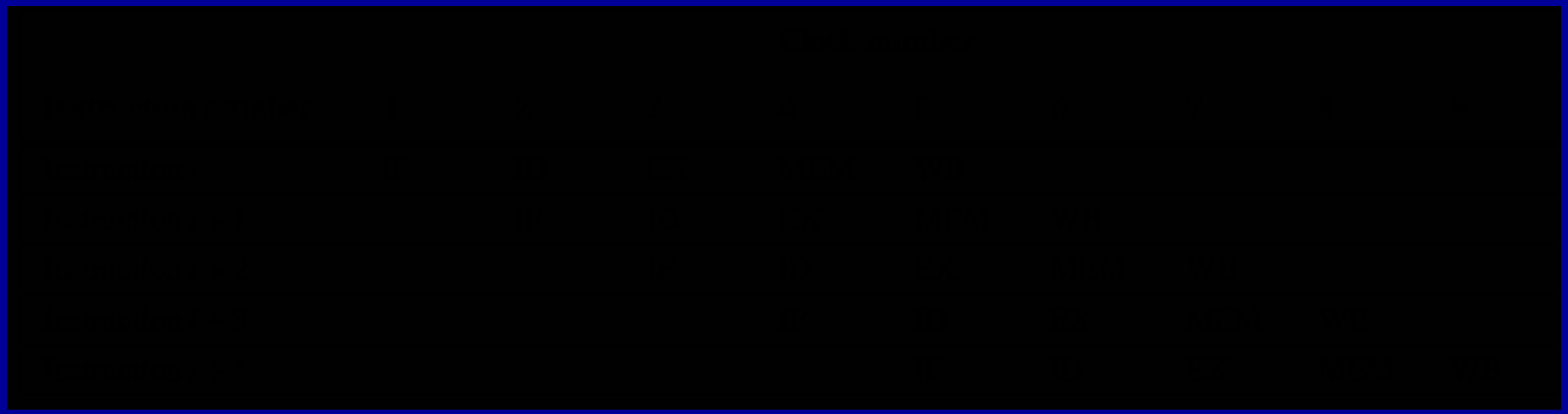


Adding Pipeline Registers

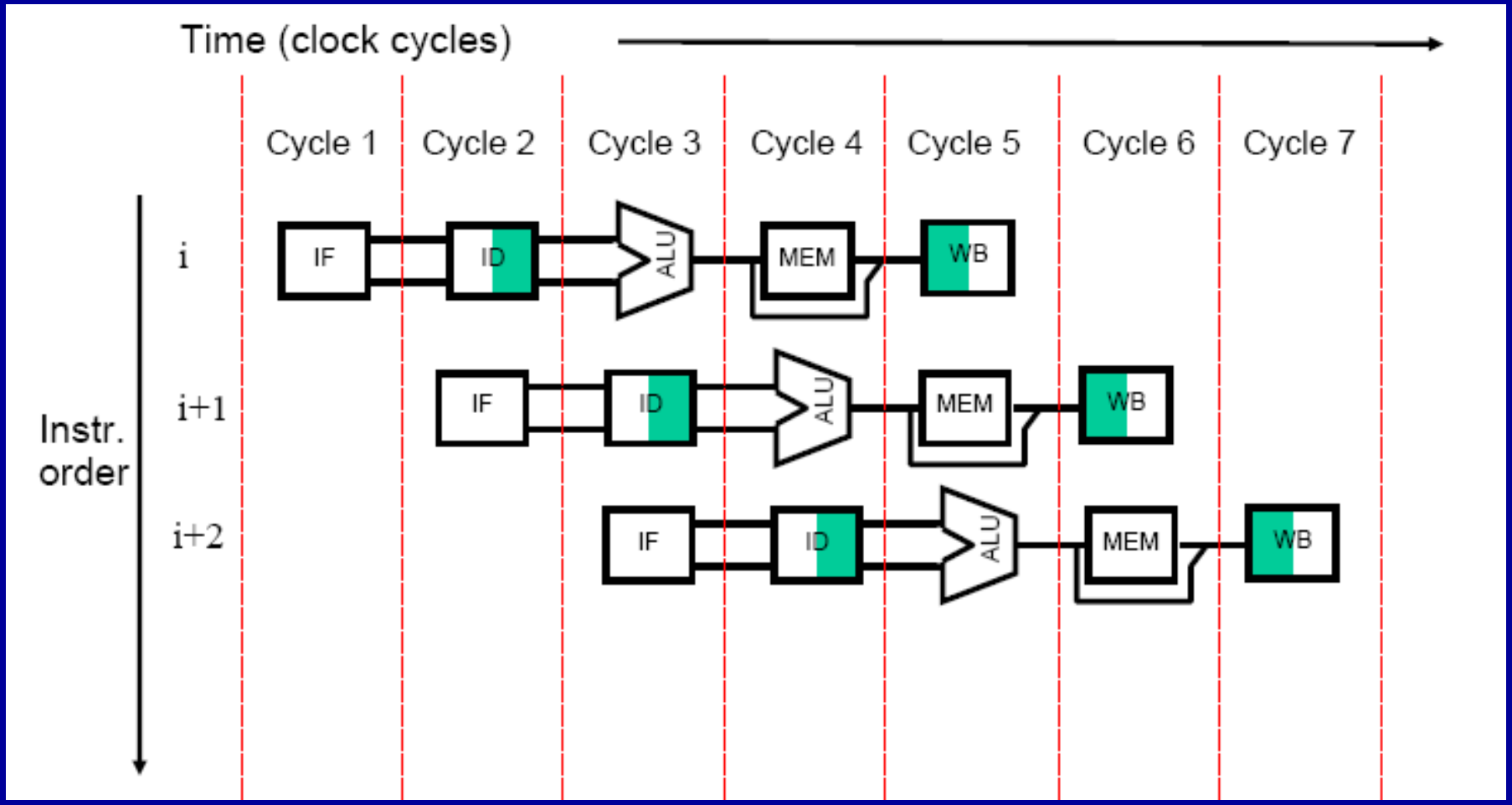


Basic RISC Pipelining

- Basic idea:
  - Each instruction spends 1 clock cycle in each of the 5 execution stages.
  - During 1 clock cycle, the pipeline can process (in different stages) 5 different instructions.



Alternative Visualization



Speedup

- Assume that a multiple cycle RISC implementation has a 10 ns clock cycle, loads take 5 clock cycles and account for 40% of the instructions, and all other instructions take 4 clock cycles.

- If pipelining the machine adds 1 ns to the clock cycle, how much speedup in instruction execution rate do we get from pipelining?

  - Average instruction execution Time (nonpipelined)
    = Clock cycle x Average CPI
    = 10 ns x (0.6 x 4 + 0.4 x 5)
    = 44 ns
  - Average instruction execution Time (pipelined)
    = 10 + 1 = 11 ns
- Speedup = 44 / 11 = 4
- The above expression assumes a pipelining CPI of 1
- Should we expect this in practice? Any complications?

Limits of Pipelining

- Increasing the number of pipeline stages in a given logic block by a factor of *k :*
  - Generally allows increasing clock speed & throughput by a factor of almost *k*.
  - Usually less than *k* because of overheads such as latches and balance of delay in each stage.
- But, pipelining has a natural limit:
  - At least 1 layer of logic gates per pipeline stage!
  - Practical minimum is usually several gates (2-10).
  - Commercial designs are rapidly nearing this point.