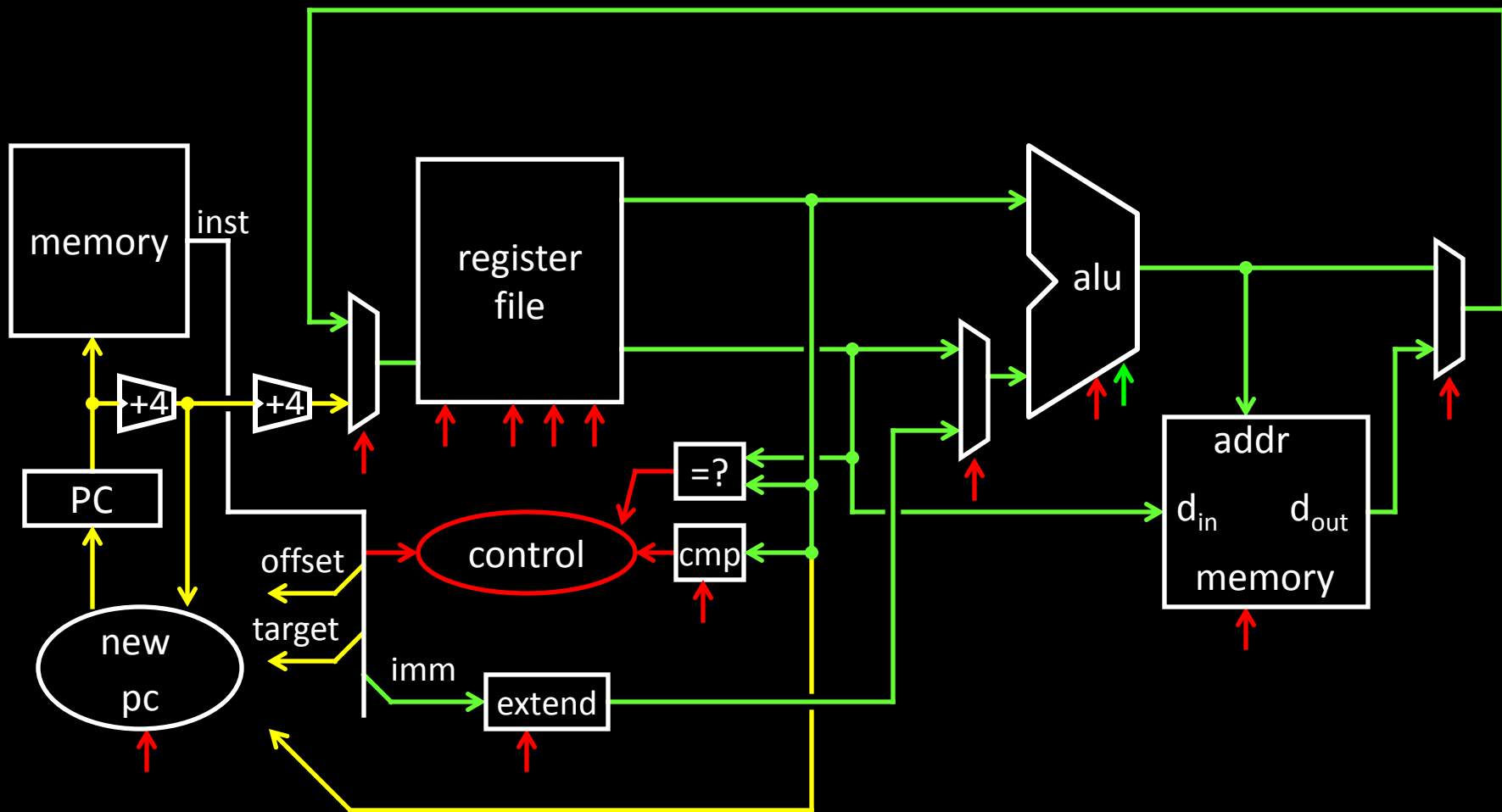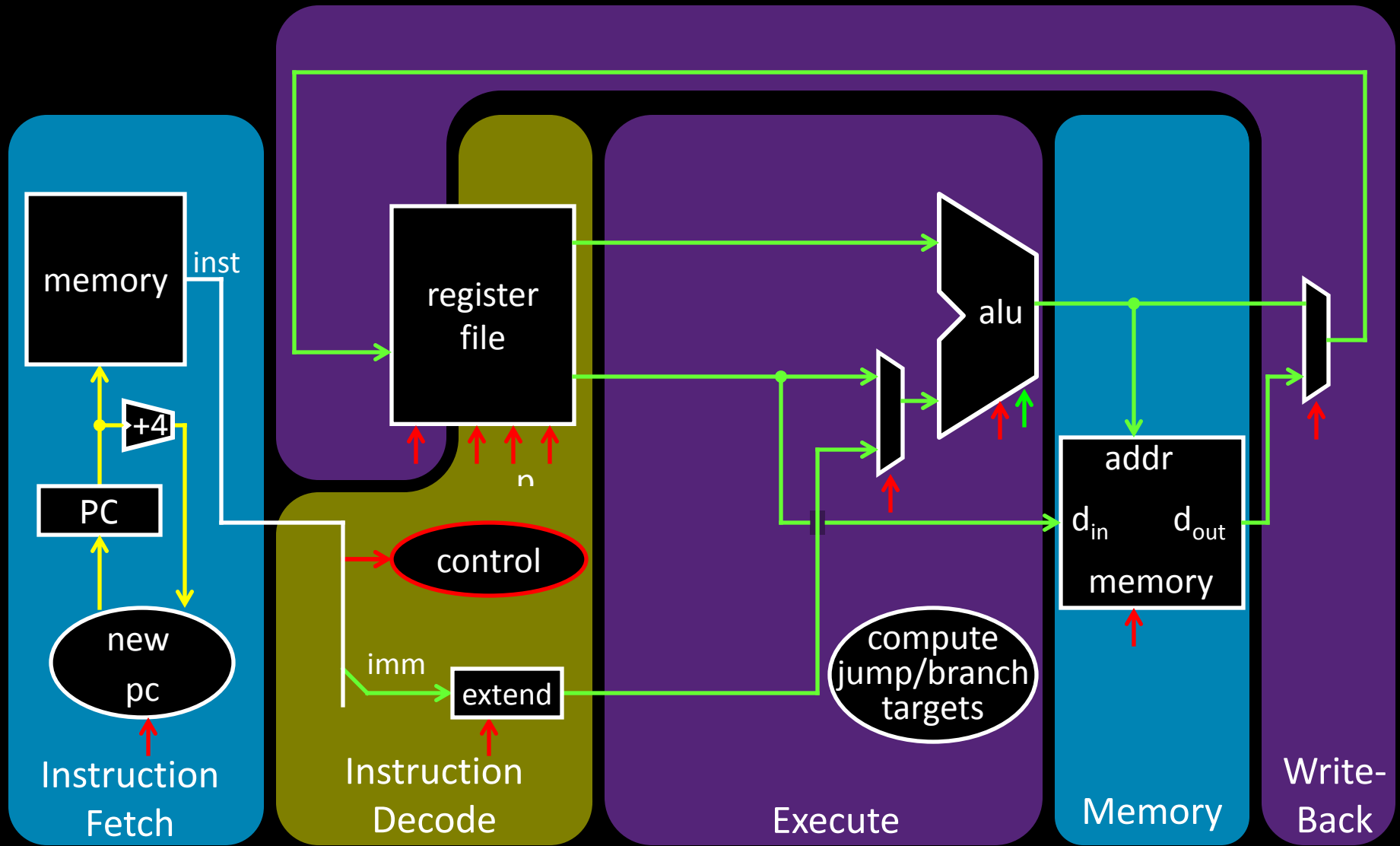# RISC Pipeline

**Kevin Walsh**
**CS 3410, Spring 2010**
Computer Science
Cornell University

See: P&H Chapter 4.6

# Five stage "RISC" load-store architecture

1. **Instruction fetch (IF)**
   - get instruction from memory, increment PC
2. **Instruction Decode (ID)**
   - translate opcode into control signals and read registers
3. **Execute (EX)**
   - perform ALU operation, compute jump/branch targets
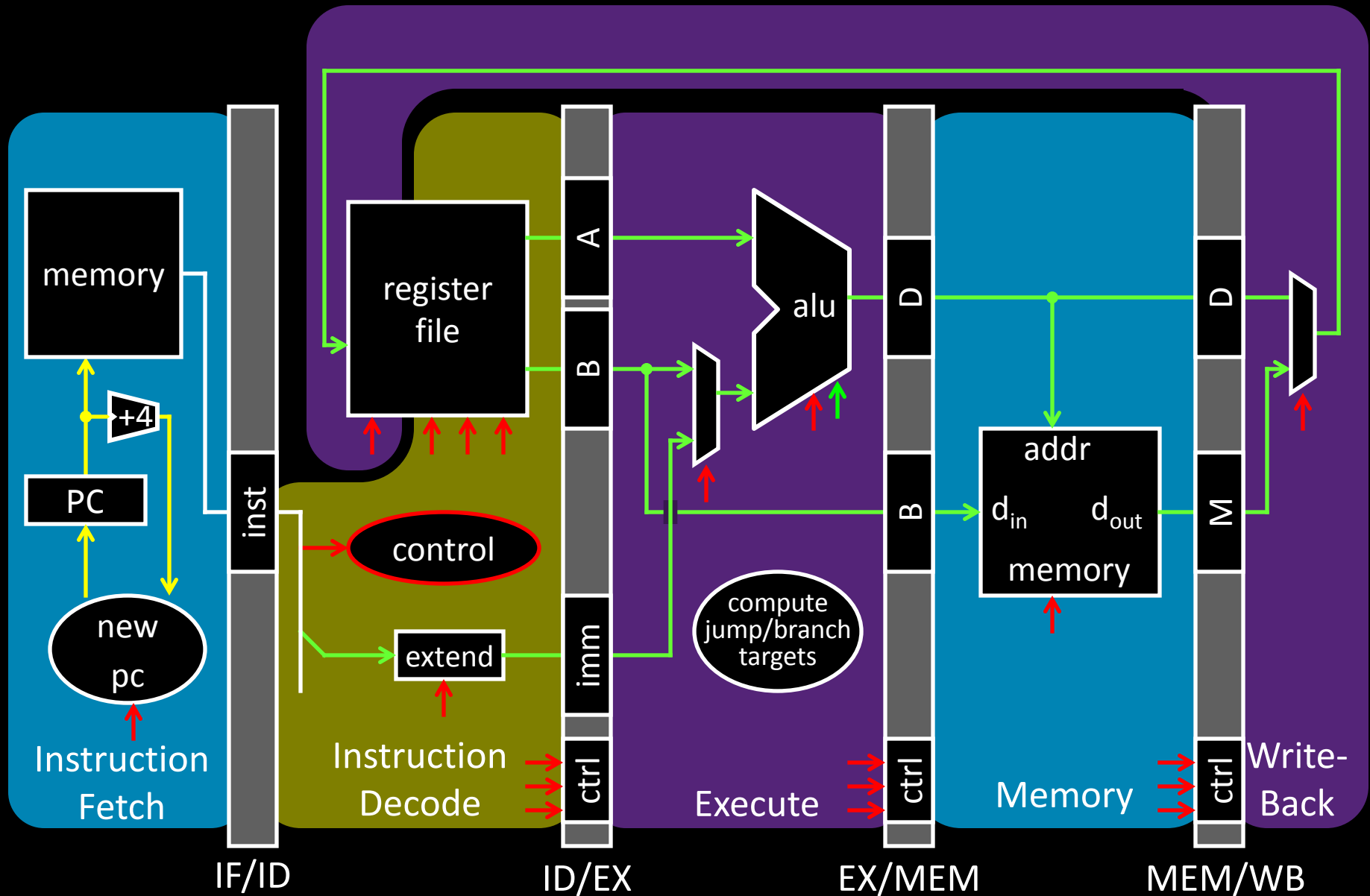4. **Memory (MEM)**
   - access memory if needed
5. **Writeback (WB)**
   - update register file

Slides thanks to Sally McKee & Kavita Bala

Break instructions across multiple clock cycles (five, in this case)

Design a separate stage for the execution performed during each clock cycle

Add pipeline registers to isolate signals between different stages
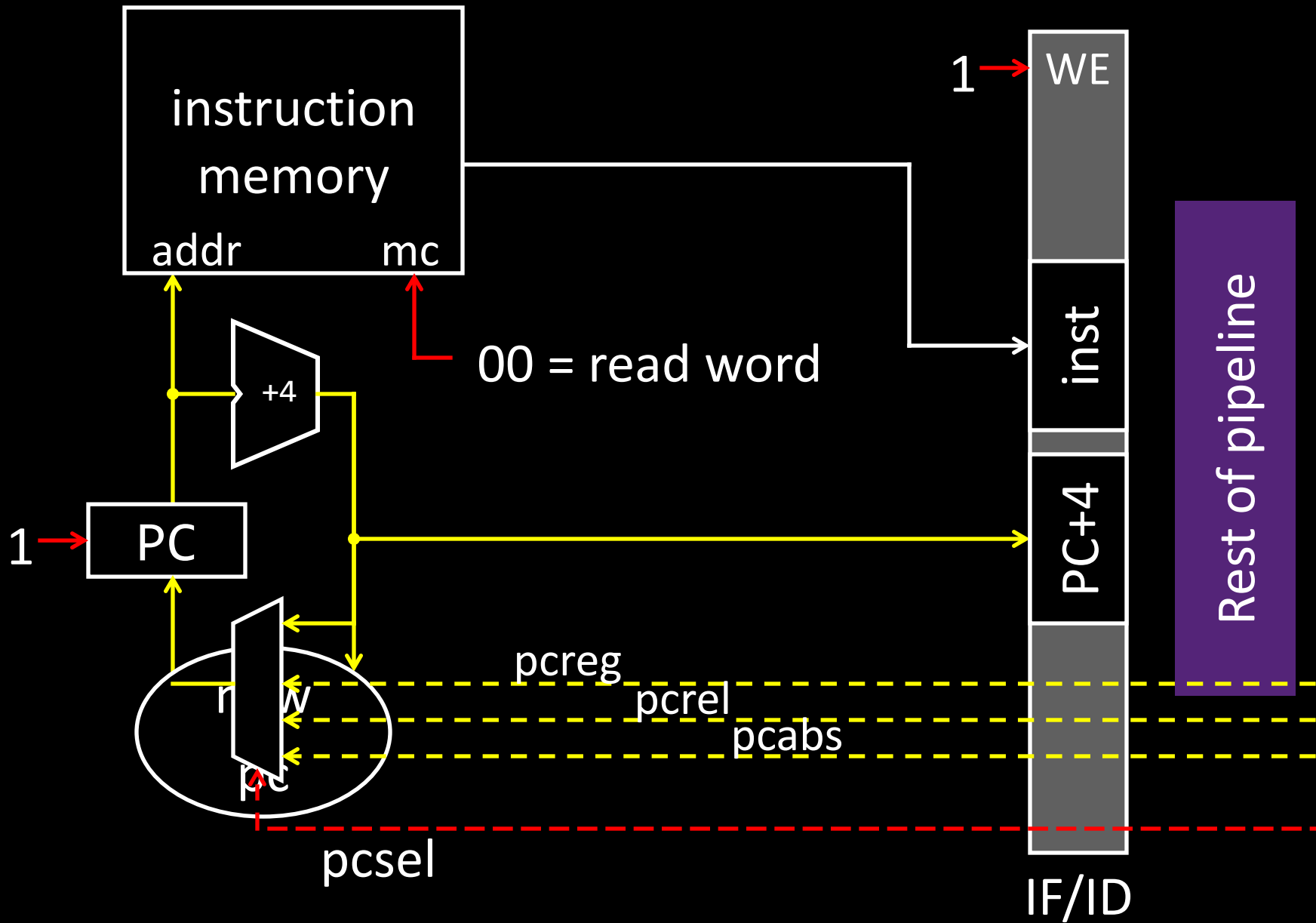
# Stage 1: Instruction Fetch

Fetch a new instruction <span style="color:red">every</span> cycle

- Current PC is index to instruction memory
- Increment the PC at end of cycle (assume no branches for now)

Write values of interest to <span style="color:red">pipeline register (IF/ID)</span>

- Instruction bits (for later decoding)
- PC+4 (for later computing branch targets)

# Stage 2: Instruction Decode

On every cycle:

- Read IF/ID pipeline register to get instruction bits
- Decode instruction, generate control signals
- Read from register file

Write values of interest to pipeline register (ID/EX)

- Control information, Rd index, immediates, offsets, …
- Contents of Ra, Rb
- PC+4 (for computing branch targets later)

dest

WE
Rd
D

register
file

A

B

Ra  Rb

decode

extend

inst

PC+4

IF/ID

A

B

imm

PC+4

ctrl
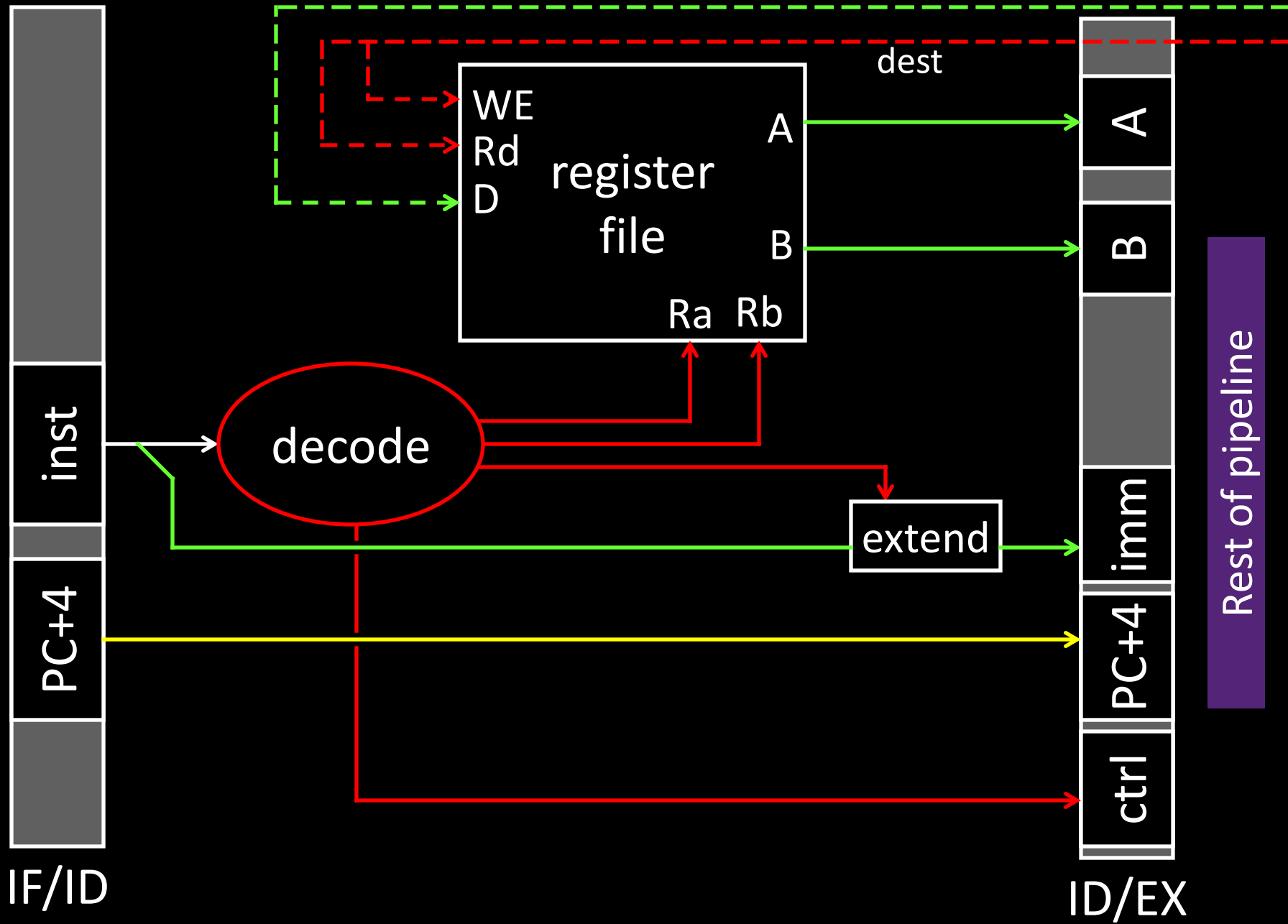
ID/EX

Rest of pipeline

# Stage 3: Execute

On every cycle:

- Read ID/EX pipeline register to get values and control bits
- Perform ALU operation
- Compute targets (PC+4+offset, etc.) *in case* this is a branch
- Decide if jump/branch should be taken

Write values of interest to pipeline register (EX/MEM)

- Control information, Rd index, …
- Result of ALU operation
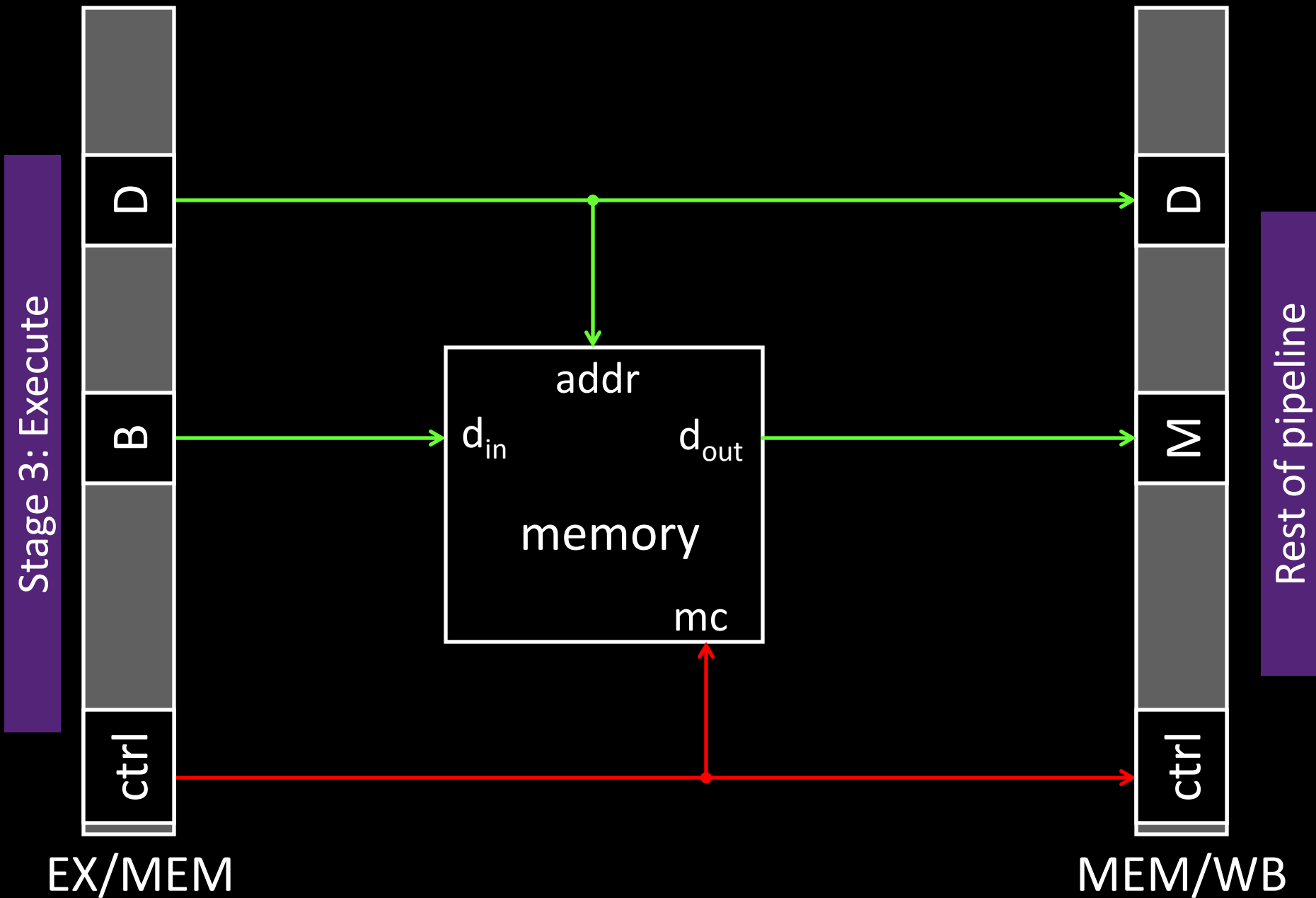- Value *in case* this is a memory store instruction
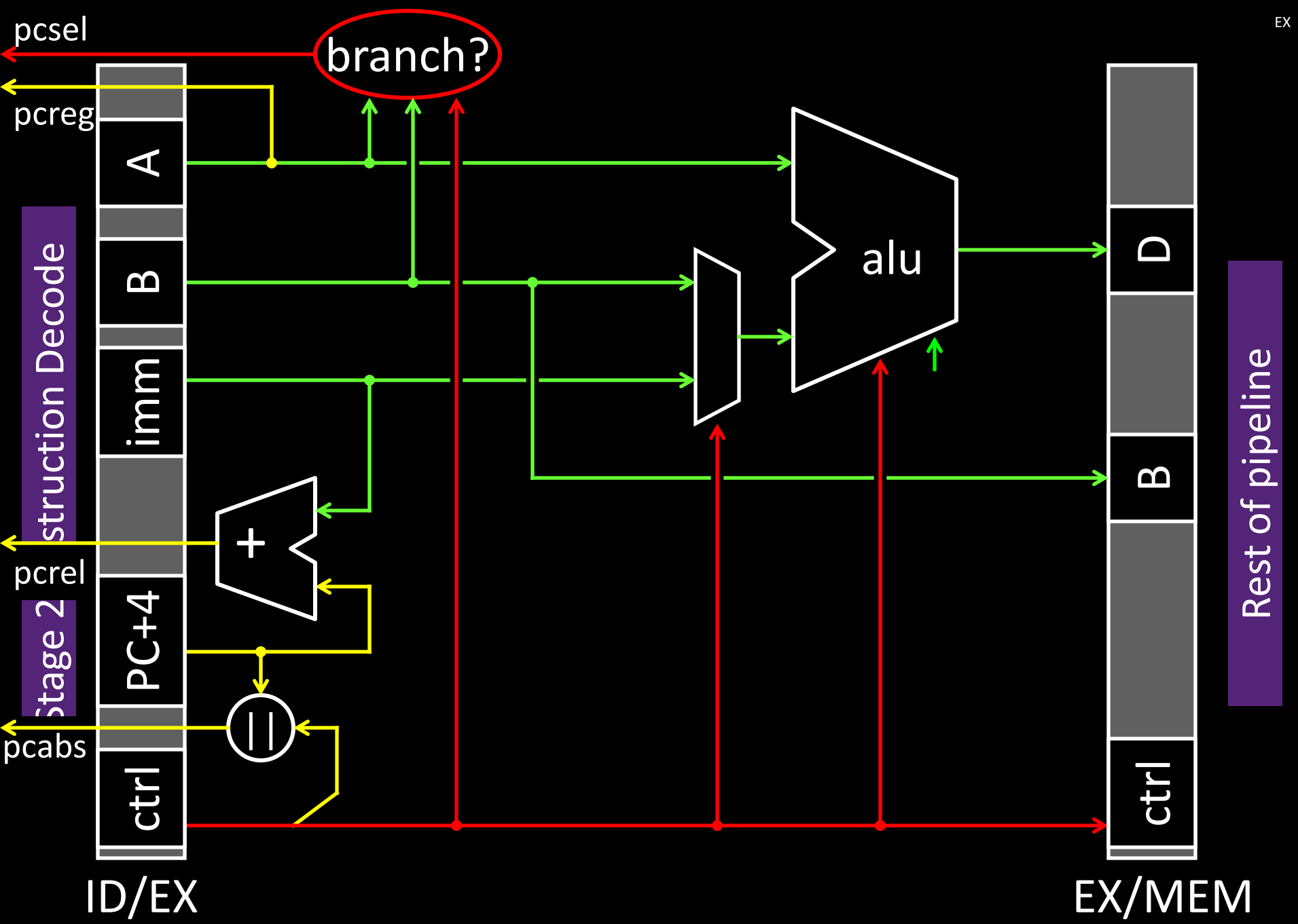
# Stage 4: Memory

On every cycle:

- Read EX/MEM pipeline register to get values and control bits
- Perform memory load/store if needed
  - address is ALU result

Write values of interest to pipeline register (MEM/WB)
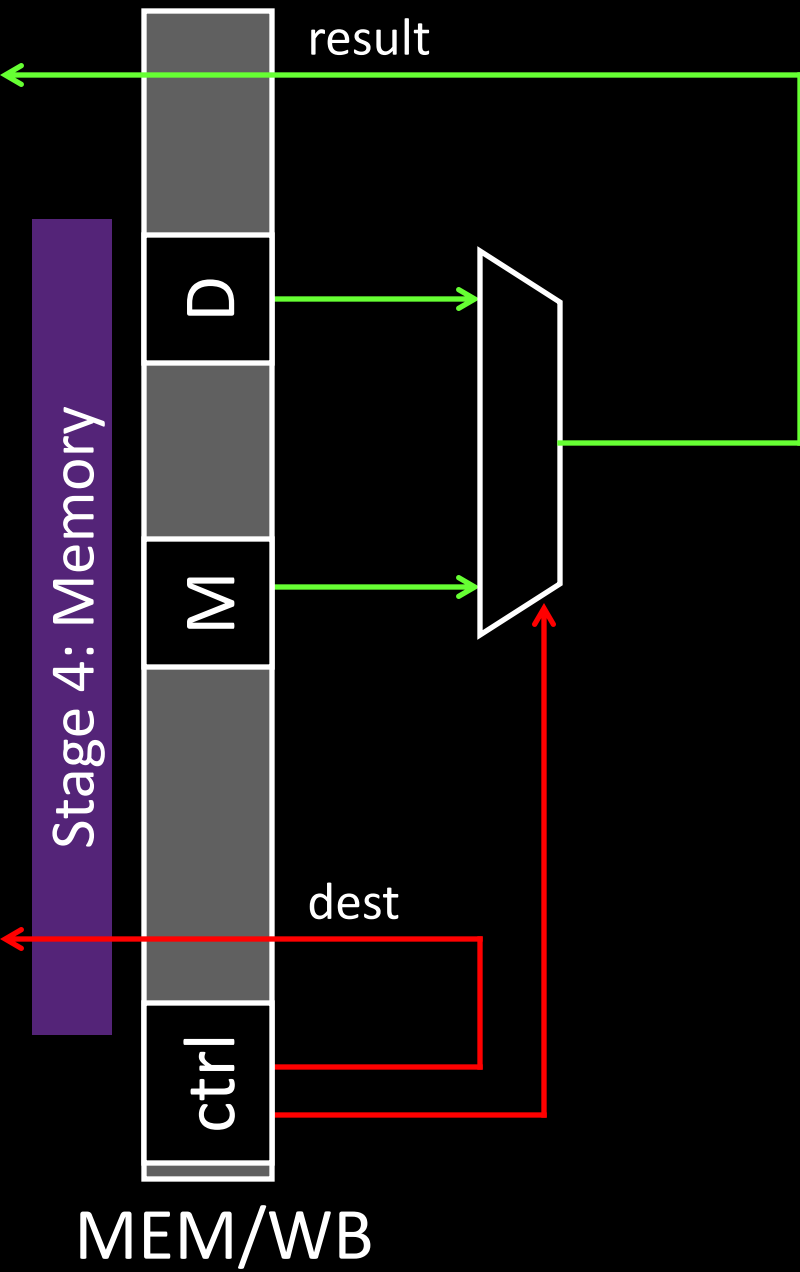
- Control information, Rd index, …
- Result of memory operation
- Pass result of ALU operation

Stage 3: Execute

Rest of pipeline

EX/MEM

MEM/WB

D

B

ctrl

D

M

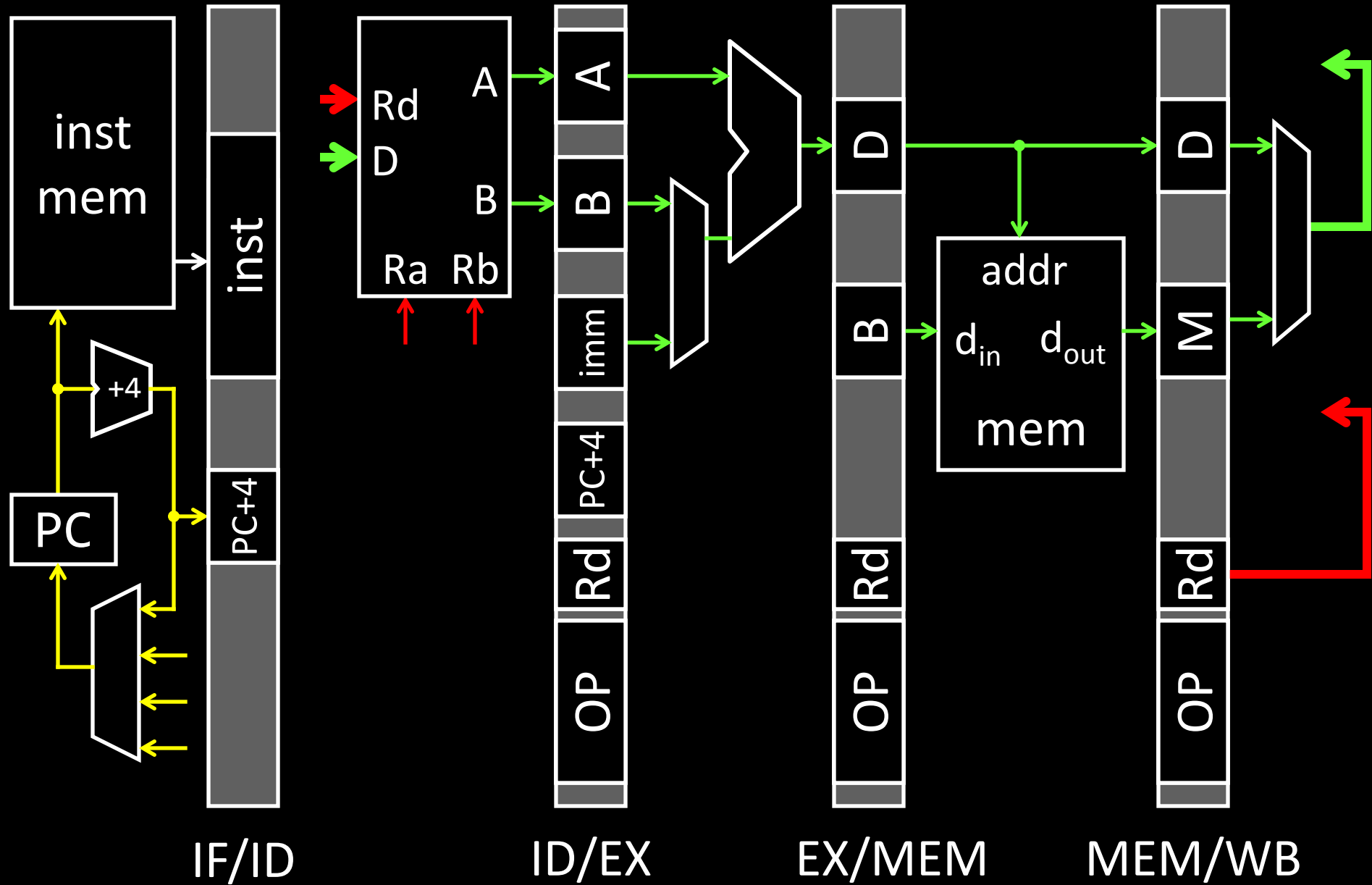ctrl

addr

$d_{in}$      $d_{out}$

memory

mc

# Stage 5: Write-back

On every cycle:

- Read MEM/WB pipeline register to get values and control bits
- Select value and write to register file

result

D

M

Stage 4: Memory

dest

ctrl

MEM/WB

inst mem

PC

+4

inst

PC+4

IF/ID

Rd
D
A
B
Ra Rb

A

B

imm

PC+4

Rd

OP

ID/EX

D

B

Rd

OP

addr

d_in    d_out

mem

EX/MEM

D

M

Rd

OP

MEM/WB

```
add   r3, r1, r2;
nand  r6, r4, r5;
lw    r4, 20(r2);
add   r5, r2, r5;
sw    r7, 12(r3);
```

inst mem

+4

PC

inst

PC+4

IF/ID

Rd

D

Ra Rb

A

B

A

B

imm

PC+4

Rd

OP

ID/EX

D

B

Rd

OP

EX/MEM

addr

d_in    d_out

mem

D

M

Rd

OP

MEM/WB

19

Clock cycle

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| add | IF | ID | EX | MEM | WB | | | | |
| nand | | IF | ID | EX | MEM | WB | | | |
| lw | | | IF | ID | EX | MEM | WB | | |
| add | | | | IF | ID | EX | MEM | WB | |
| sw | | | | | IF | ID | EX | MEM | WB |
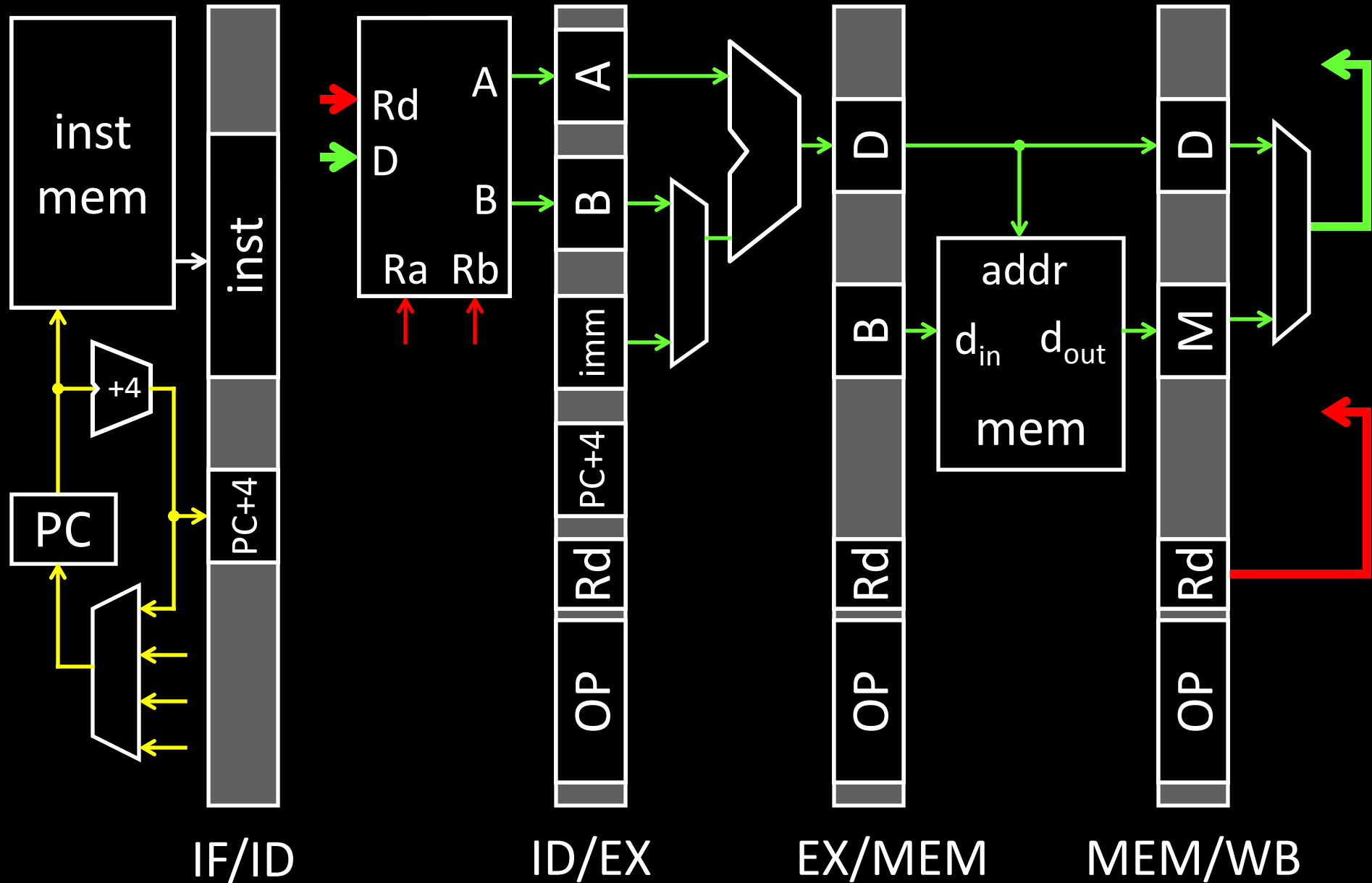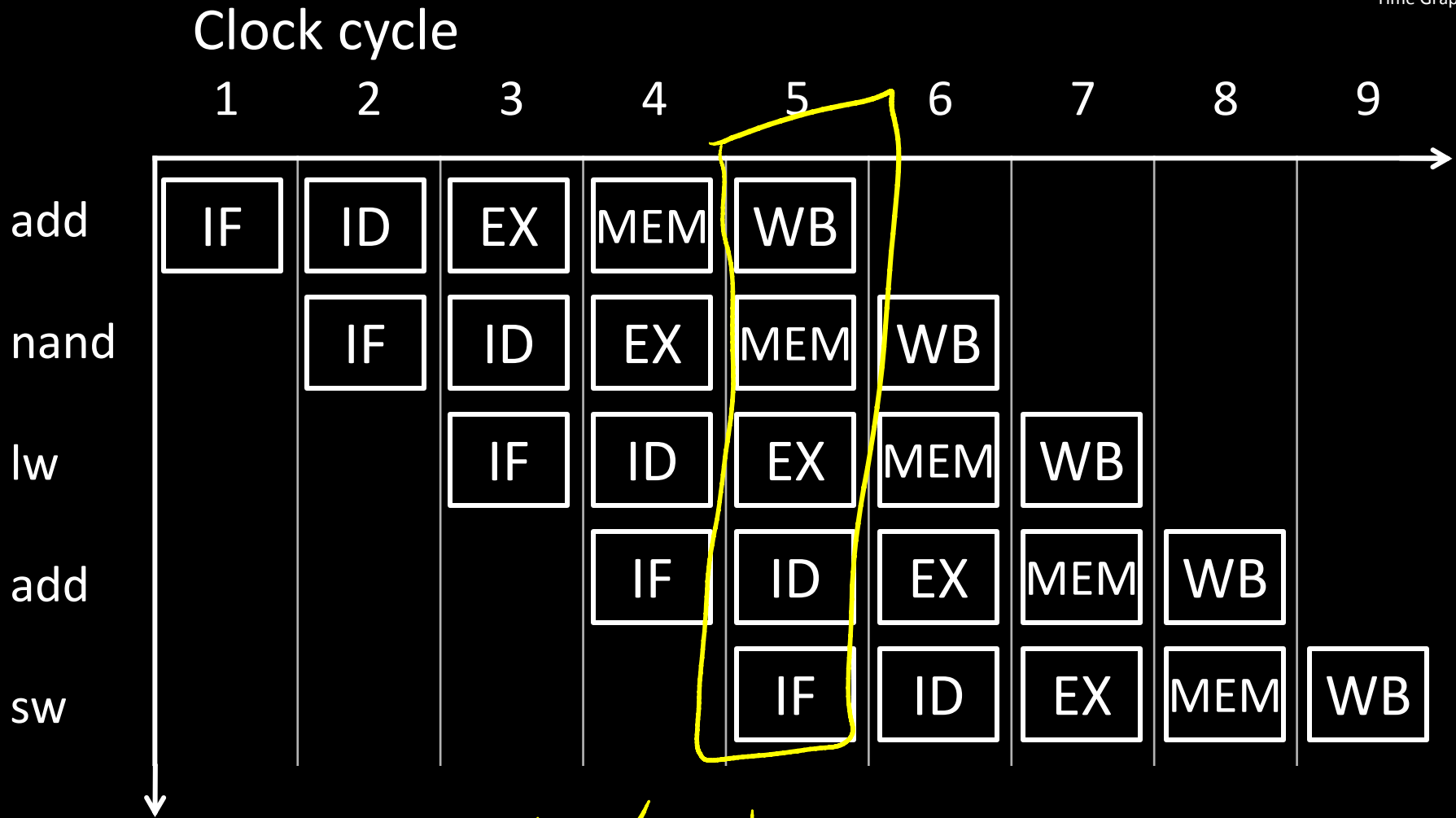
Latency: 5 cycles / inst.

Throughput: 1 inst / cycle

Concurrency: 5

CPI = ~ 1.0

# Powerful technique for masking latencies

- Logically, instructions execute one at a time
- Physically, instructions execute in parallel
  - Instruction level parallelism

# Abstraction promotes decoupling

- Interface (ISA) vs. implementation (Pipeline)