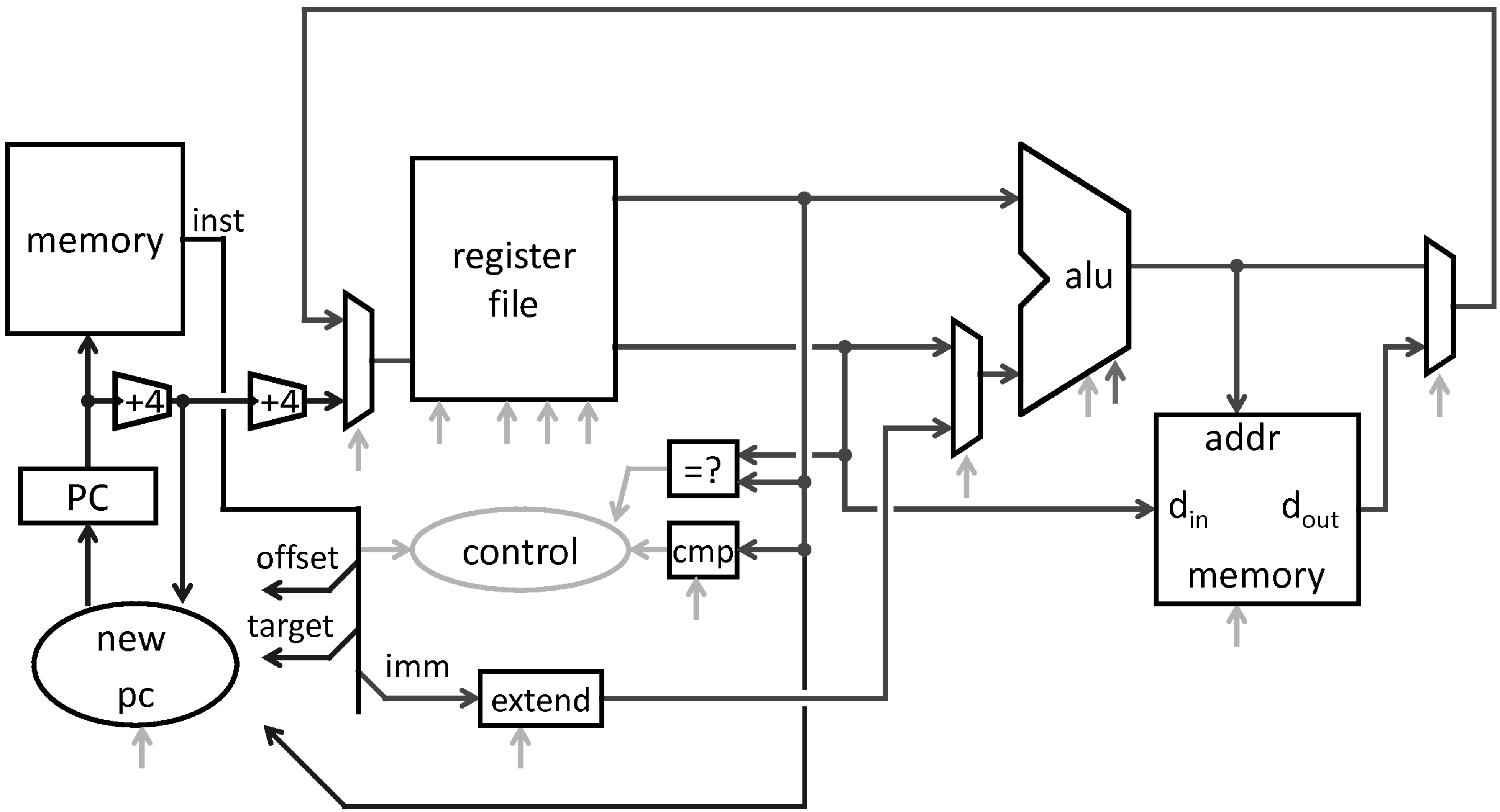
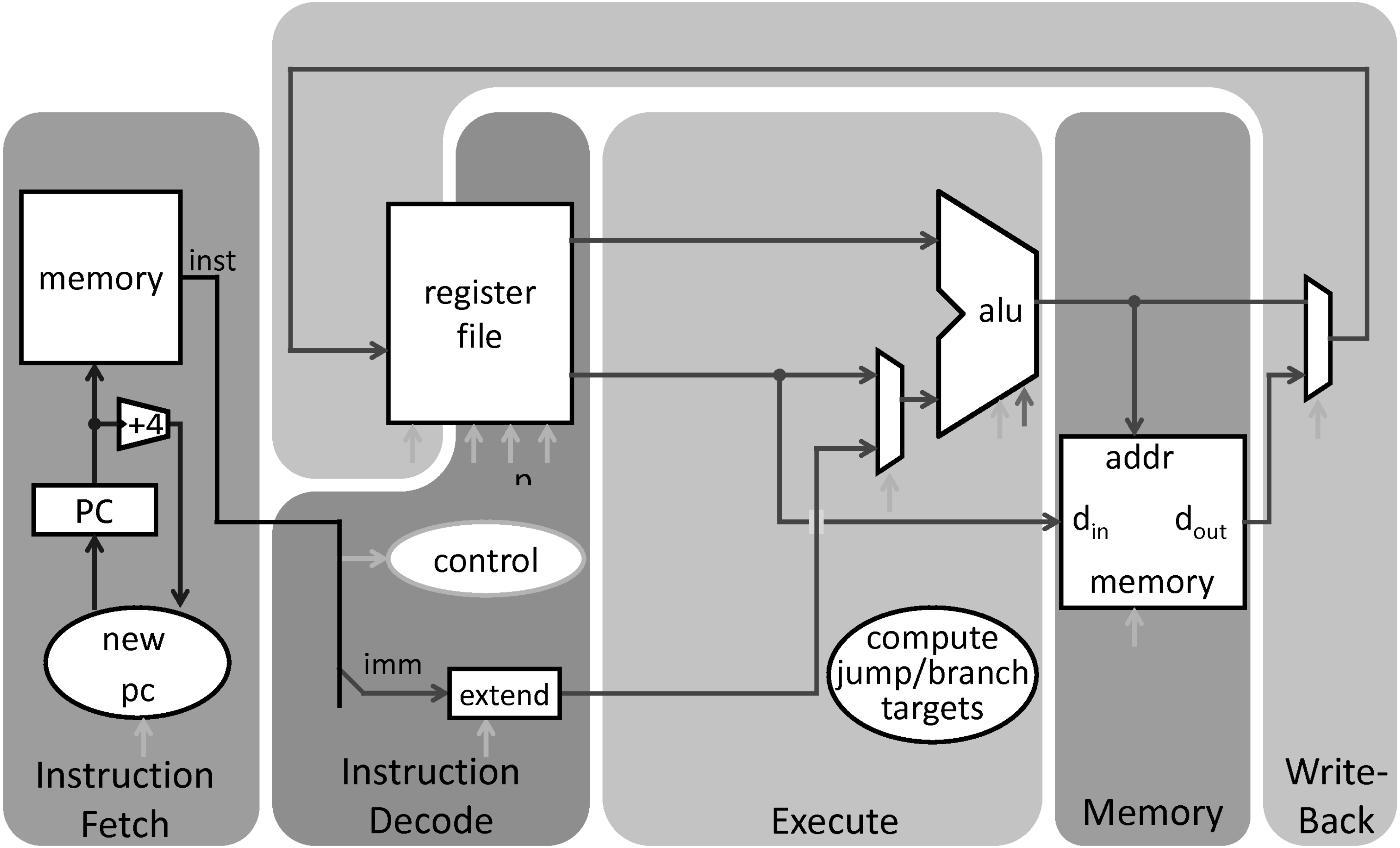
RISC Pipeline

Kevin Walsh
CS 3410, Spring 2010
Computer Science
Cornell University

See: P&H Chapter 4.6


memory
inst
register
file
alu
+4
+4
PC
=?
control
cmp
offset
target
imm
extend
new
pc
addr
d_{in}
d_{out}
memory


memory
inst
register
file
alu
+4
PC
new
pc
control
imm
extend
addr
d_{in}
d_{out}
memory
compute
jump/branch
targets
Instruction
Fetch
Instruction
Decode
Execute
Memory
Write-
Back

Five stage "RISC" load-store architecture

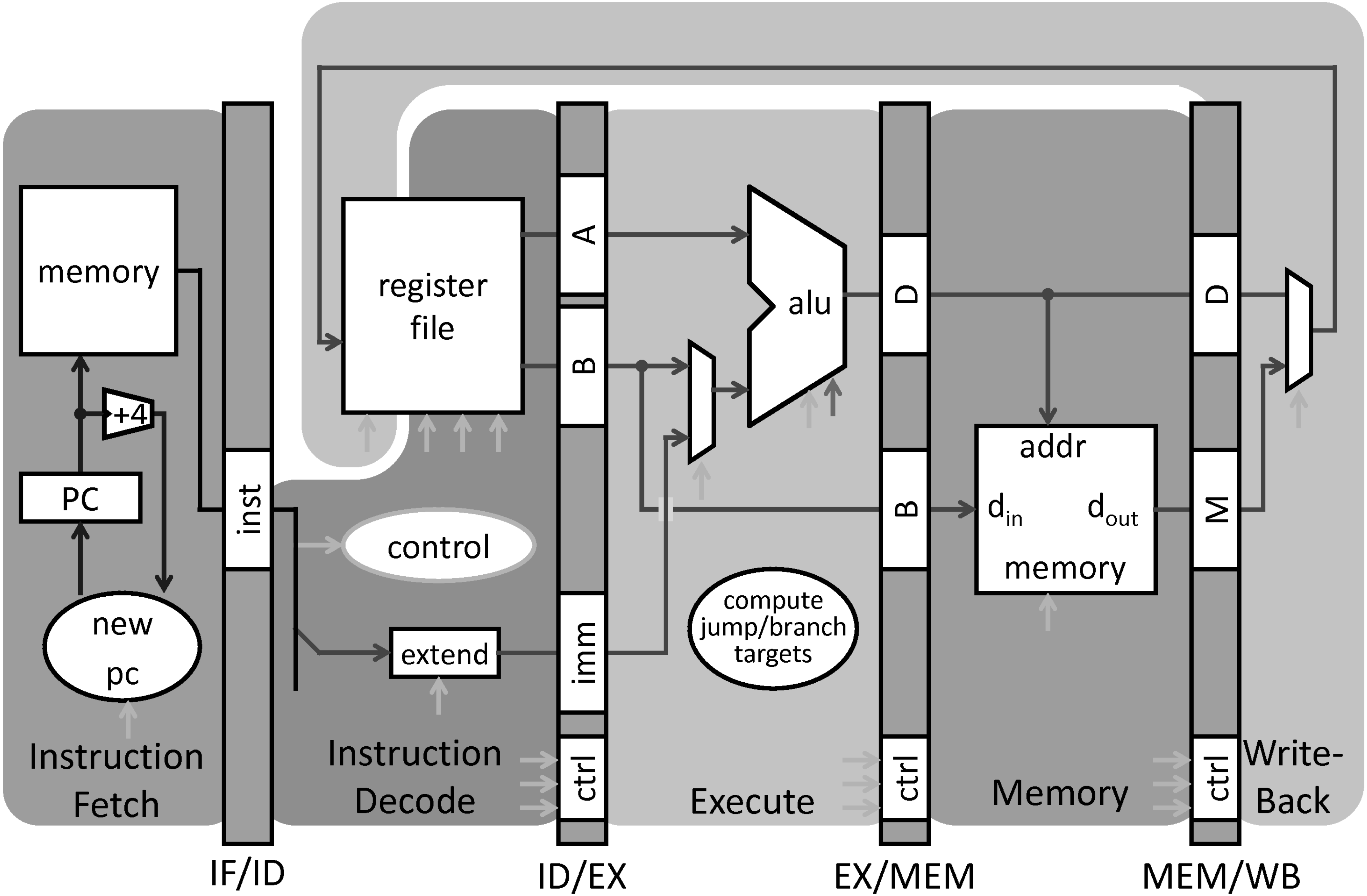
1. Instruction fetch (IF)
   – get instruction from memory, increment PC
2. Instruction Decode (ID)
   – translate opcode into control signals and read registers
3. Execute (EX)
   – perform ALU operation, compute jump/branch targets
4. Memory (MEM)
   – access memory if needed
5. Writeback (WB)
   –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


memory
PC
+4
new pc
Instruction Fetch
inst
IF/ID
register file
control
extend
Instruction Decode
A
B
imm
ctrl
ID/EX
alu
compute jump/branch targets
Execute
D
B
ctrl
EX/MEM
addr
d_{in}
d_{out}
memory
Memory
D
M
ctrl
MEM/WB
Write-Back

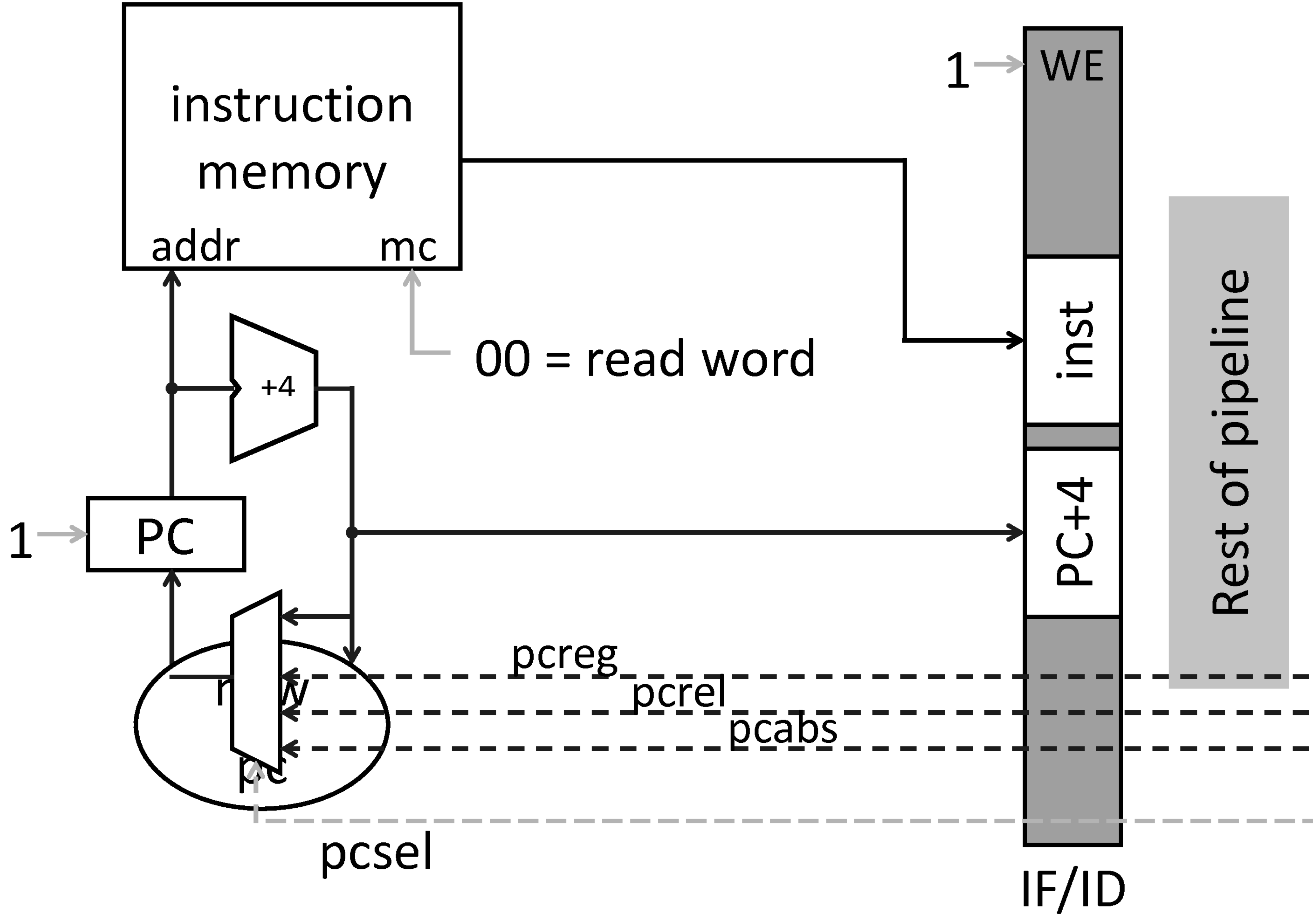
Stage 1: Instruction Fetch

Fetch a new instruction every 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 pipeline register (IF/ID)

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


instruction
memory
addr
mc
00 = read word
+4
1
PC
WE
inst
PC+4
Rest of pipeline
pcreg
pcrel
pcabs
pcsel
IF/ID

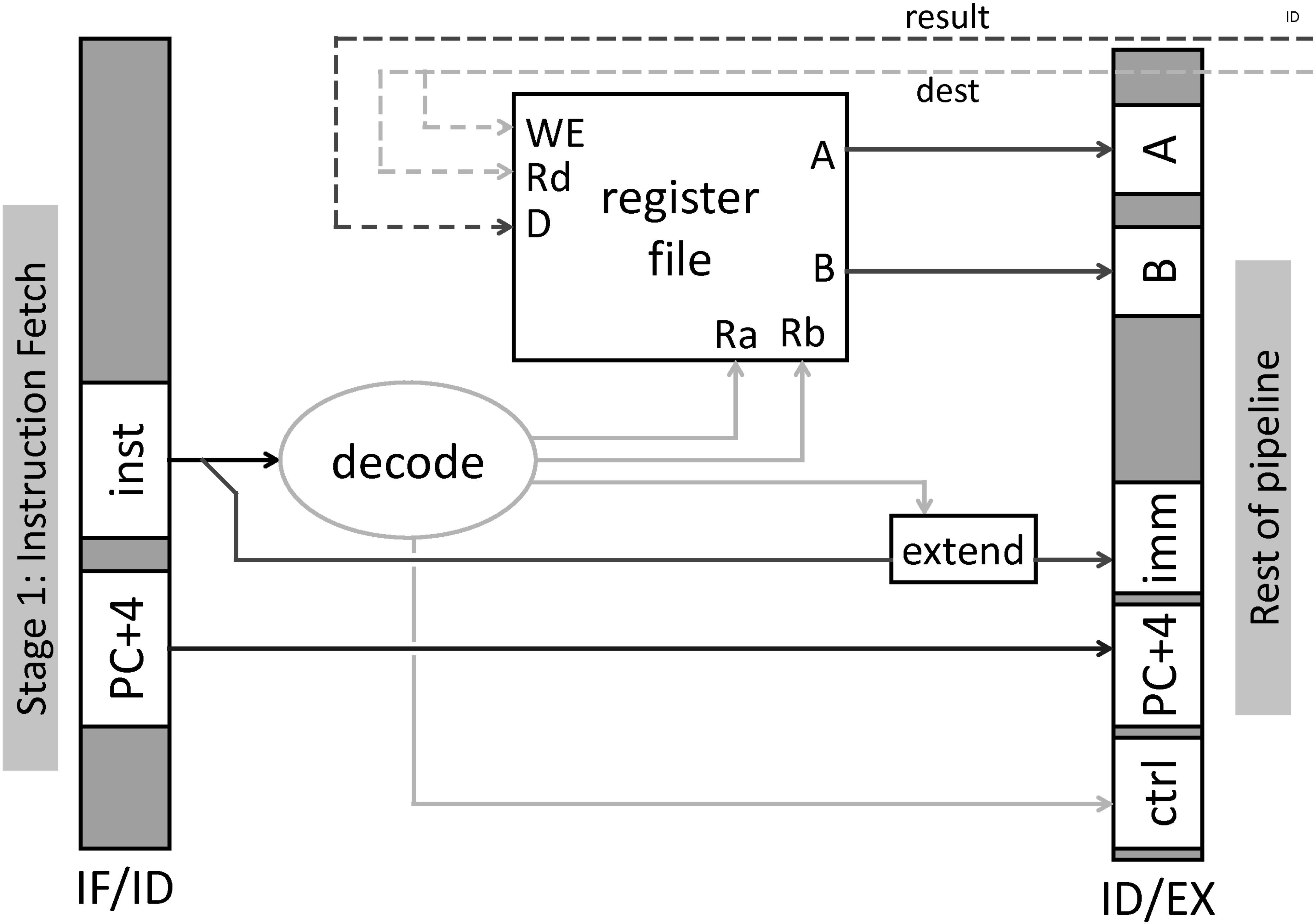
Stage 2: Instruction Decode

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)


result
dest
WE
Rd
D
register
file
A
B
Ra
Rb
decode
extend
Stage 1: Instruction Fetch
inst
PC+4
IF/ID
A
B
imm
PC+4
ctrl
ID/EX
Rest of pipeline

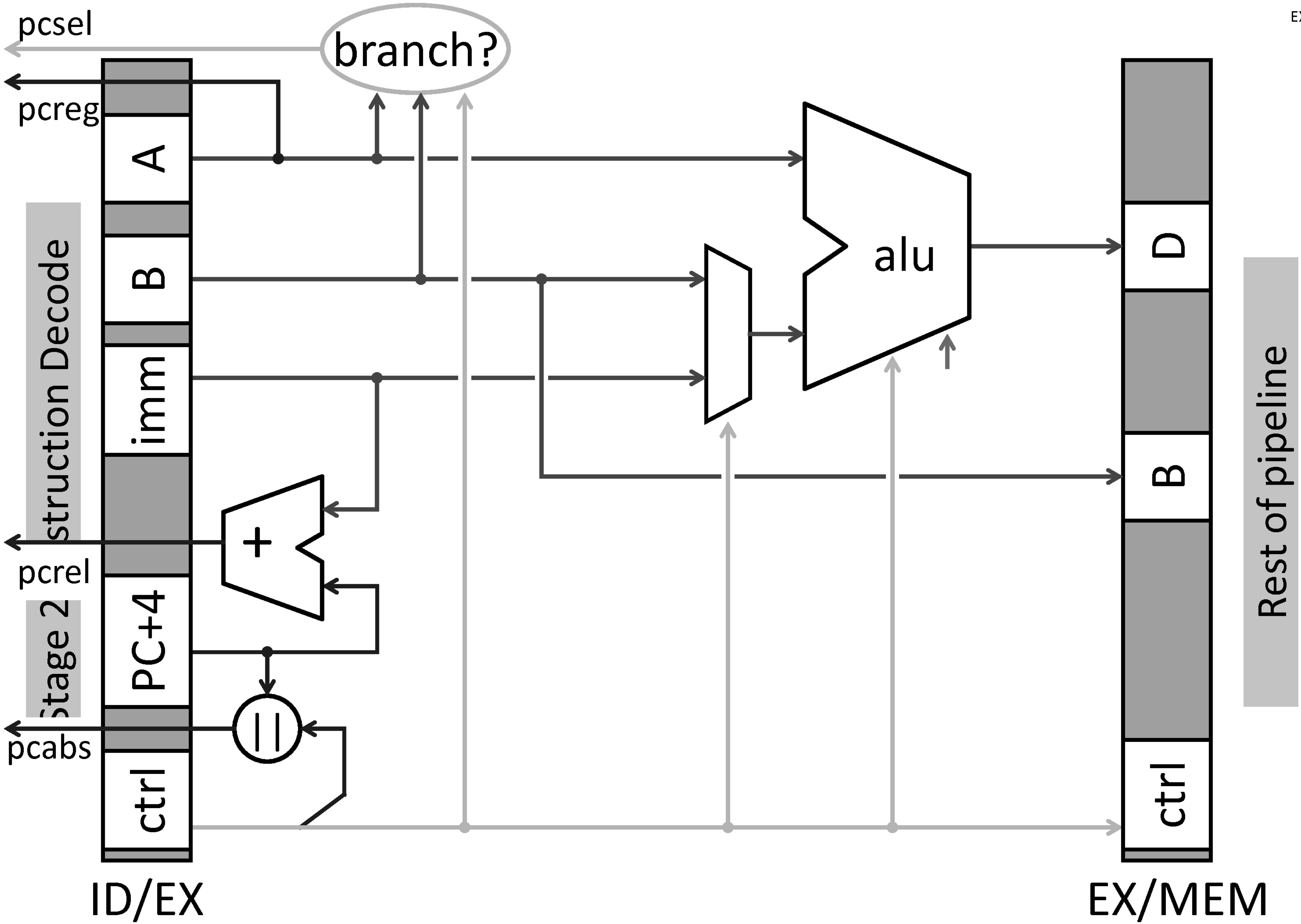
Stage 3: Execute

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


pcsel
branch?
pcreg
A
B
imme
alu
D
B
+
pcrel
PC+4
||
pcabs
ctrl
ctrl
Stage 2
Instruction Decode
Rest of pipeline
ID/EX
EX/MEM

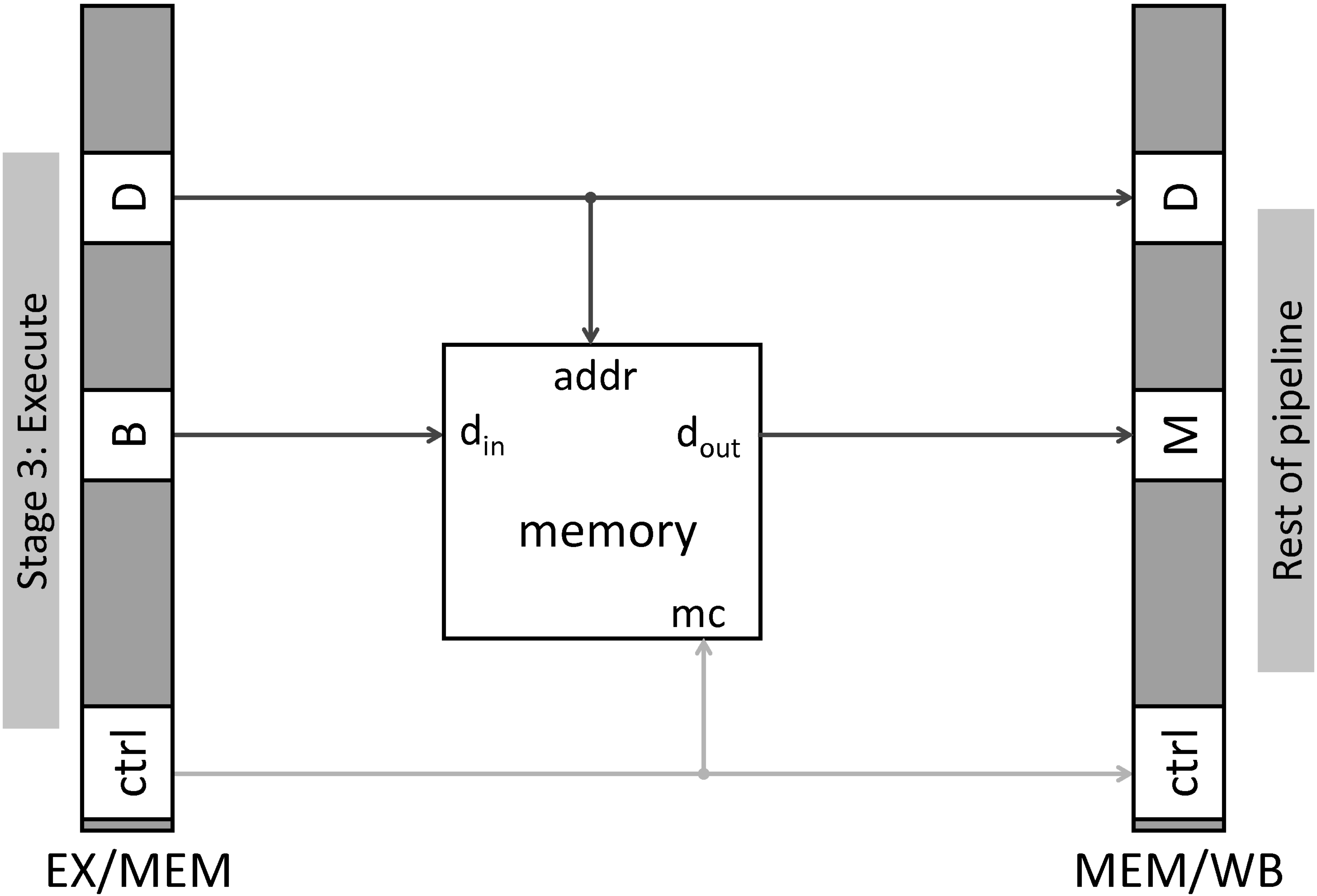
Stage 4: Memory

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)

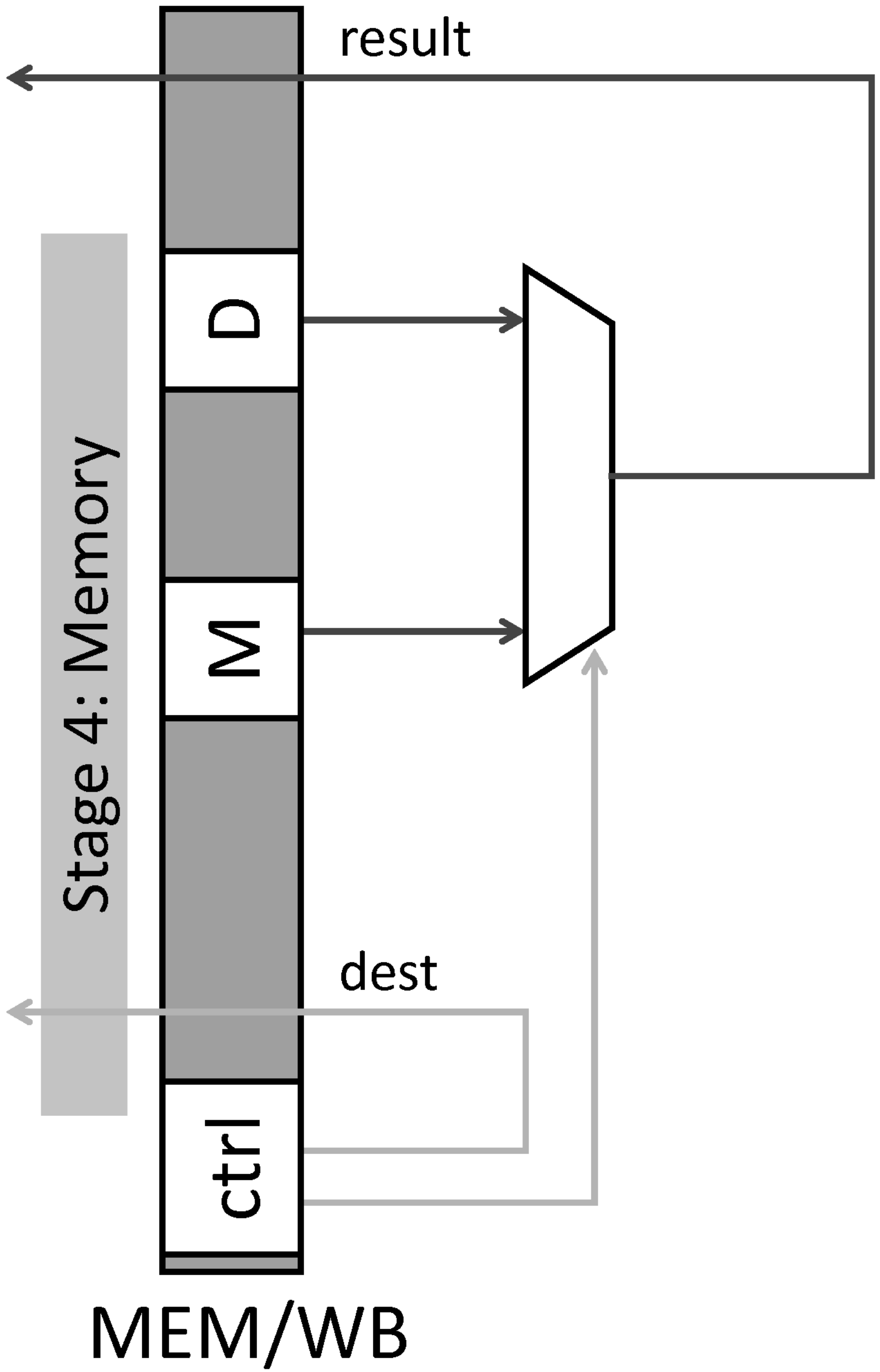
- Control information, Rd index, …
- Result of memory operation
- Pass result of ALU operation

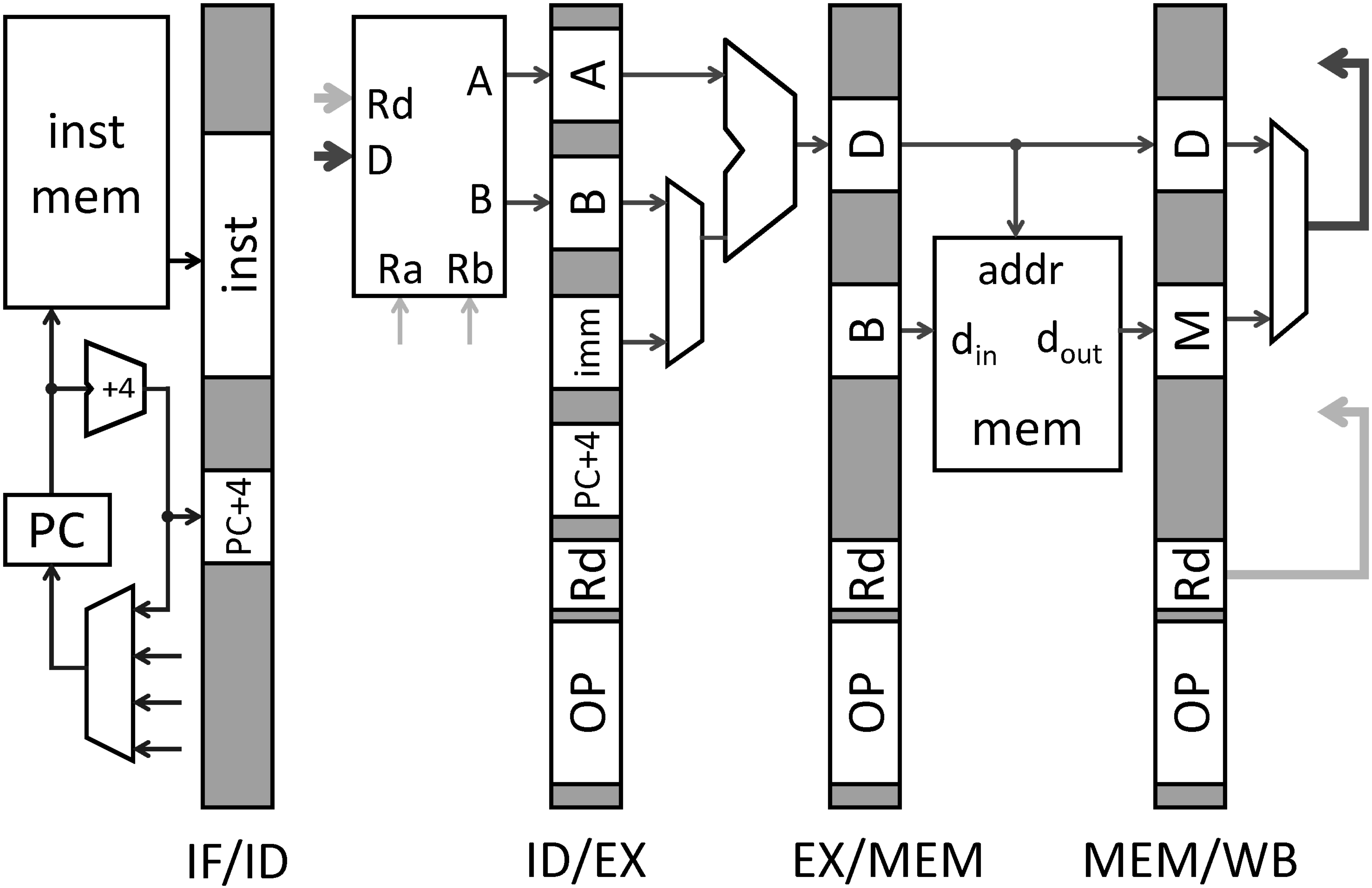

Stage 3: Execute
D
B
ctrl
addr
d_in
d_out
memory
mc
D
M
ctrl
Rest of pipeline
EX/MEM
MEM/WB

Stage 5: Write-back

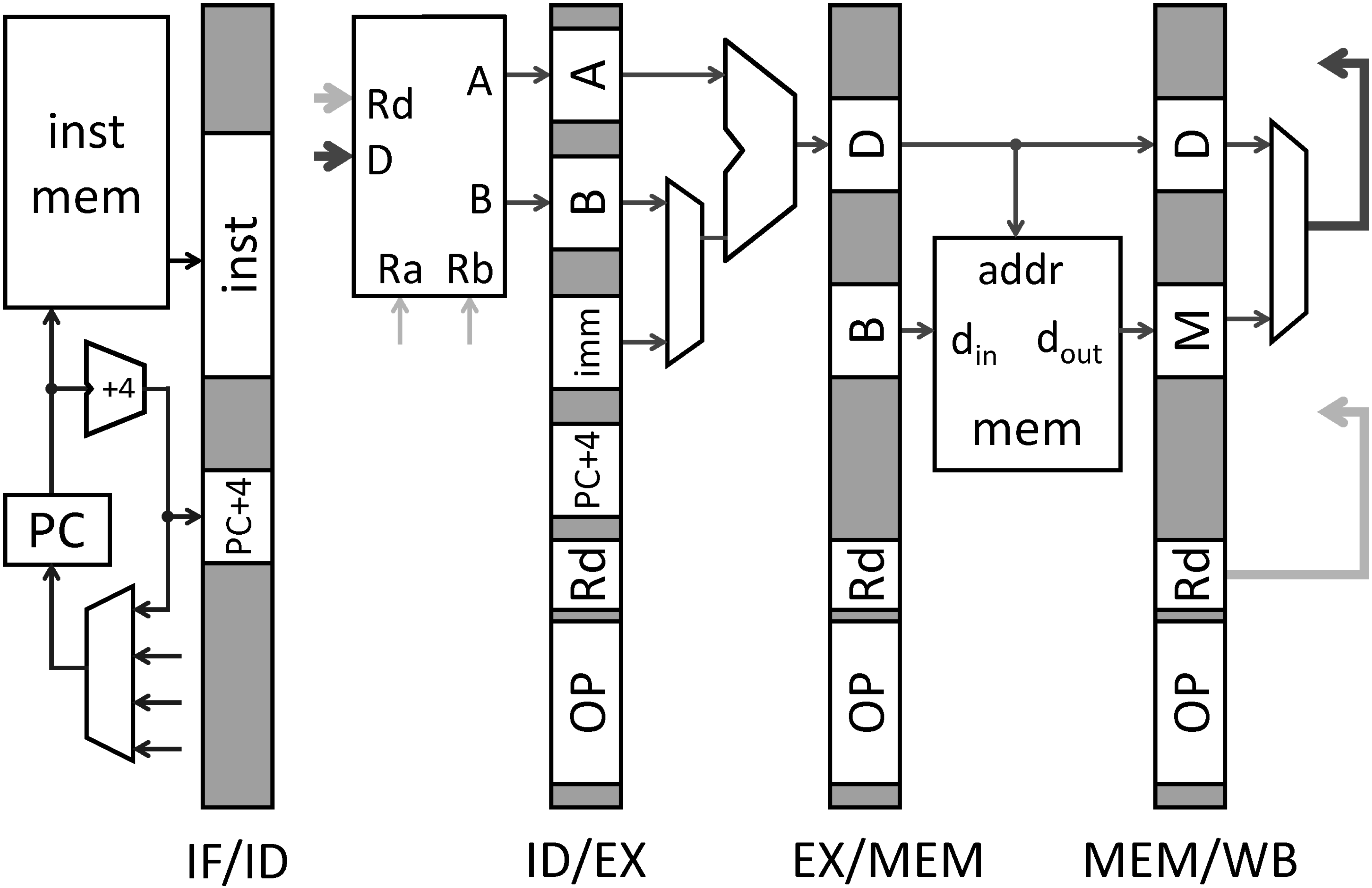
On every cycle:

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


result
Stage 4: Memory
D
M
dest
ctrl
MEM/WB


inst
mem
PC
+4
inst
PC+4
Rd
D
A
B
Ra
Rb
A
B
imm
PC+4
Rd
OP
D
B
Rd
OP
addr
d_{in}
d_{out}
mem
D
M
Rd
OP
IF/ID
ID/EX
EX/MEM
MEM/WB

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


inst
mem
PC
+4
inst
PC+4
Rd
D
A
B
Ra
Rb
A
B
imm
PC+4
Rd
OP
D
B
Rd
OP
addr
d_in
d_out
mem
D
M
Rd
OP
IF/ID
ID/EX
EX/MEM
MEM/WB

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:

Throughput:

Concurrency:

CPI =

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)