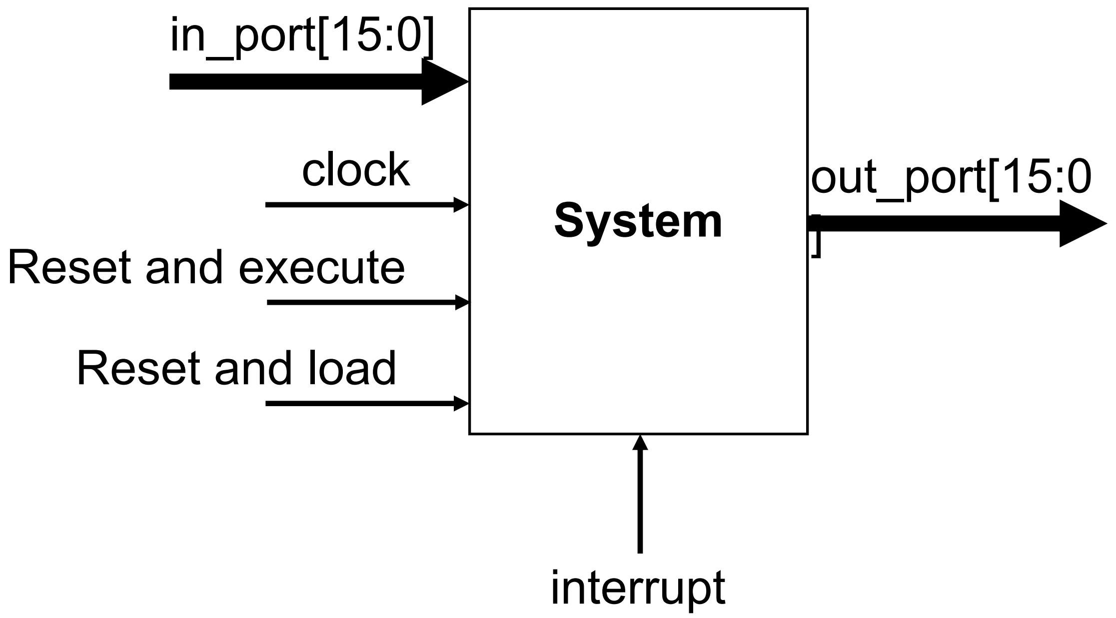
ECE 449 Project

University of Victoria

Pinout of Processor

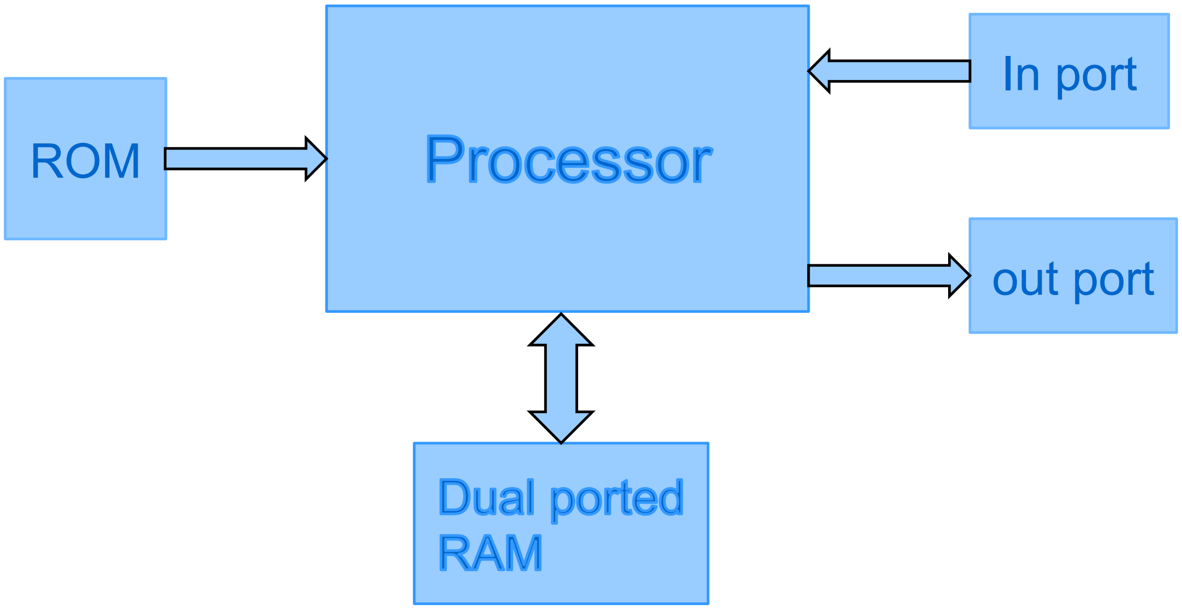
- Interrupt is optional



Resets

- Reset and execute causes the system to execute the user's code
- Reset and load causes the system to load the user's code into RAM
- Explanations and detailed specifications will follow in slide 7

System Description



Comments on the system architecture -RAM

- A dual ported RAM is used to ensure that instruction and data traffic is separated (Harvard architecture)
- The RAM serializes the access requests arbitrarily.
- The suggested (slide 9) RAM implementation is a synchronous one and it allows the specification of the delay (in clock cycles) of the read data to appear on the data_out port.

Comments on the system architecture -ROM

- ROM is used to store a rudimental BIOS.
- ROM and its BIOS will be provided to you
- The main functionality of the BIOS is
  - Load user code into the appropriate location in RAM
  - Execute user code

Resets

- The two resets implement the load and execute functionality of the BIOS
- Both clear the PC
- **Reset and Execute** *vectors* to address 0x0000 while **Reset and Load** *vectors* to address 0x0002.
- At each address, the developer has introduced the appropriate branch (BRR) instruction that vectors to the reset-handling routine (this is part of the BIOS)

ROM, RAM and ports

- ROM is 1024-byte large starting at address 0x0000
- RAM is a 1024 byte block starting at address 0x0400
- We use memory-mapped ports. The input port is located at 0xFFF0 while the output port is at 0xFFF1

RAM module

- Please use the dual port distributed RAM macro XPM_MEMORY_DPDISTRAM from Xilinx XPM macro group
- This macro can configure a dual ported memory where port A can be used for both reading and writing while port B can only be used for reading only (from memory).

Instruction Format

- Three types of instructions
  - A-Format
    - e.g. arithmetic instructions
  - B-Format
    - e.g. branch instructions
  - L-Format
    - e.g. load and store instructions

A-Format Instructions

Arithmetic Instructions:

15 … 9	8 … 0
op-code	xxxx[1]

Format A0

15 … 9	8 … 6	5 … 3	2 … 0
op-code	ra	rb	rc

Format A1

15 … 9	8 … 6		3 … 0
op-code	ra	xxxxx	c1

Format A2

15 … 9	8 … 6	5 … 0
op-code	ra	xxxx

Format A3

e.g.:

ADD r3,r2,r1

15 … 9	8 … 6	5 … 3	2 … 0
0000001	011	010	001

B-Format Instructions

Branch Instructions:

15 ... 9	8 ... 0
op-code (BRR)	disp.l

Format B1

15 ... 9	8 ... 6	5 ... 3 2 ... 0
op-code (BR)	ra	disp.s

Format B2

e.g.:

BR r3+0x8

15 ... 9	8 ... 6	5 ... 0
1000011	**011**	**001000**

Subroutine

…

…

BR.SUB r2+072

…

subroutine:

…

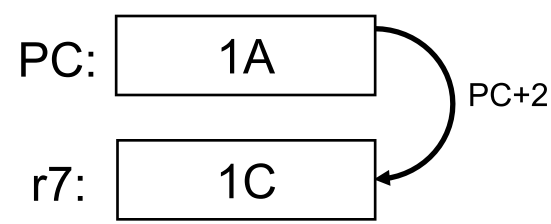
return

Subroutine

- Register r7 plays the additional role of a Link Register used in subroutine call&return
  - **BR.SUB rx+disp**

The current program counter is placed in r7 while the PC is loaded with the target address (i.e. (rx)+2*disp)

```
           …
           …
0X1A:  BR.SUB
0X1C:  …
subroutine:
           …
           return
```



For the PC operations, the argument is the value of the PC just before the instruction is fetched, while the result is the value of the PC at the conclusion of the execution of the instruction.

Subroutine

- LR: a dedicated register for br.sub instructions
  - br.sub: PC+2 is loaded into r7
  - return: PC is loaded with r7

	...	PC:	8A
	...		
0X1A:	**BR.SUB**	r7:	1C
0X1C:	...		
subroutine:			
	...		
0X88:	**return**		

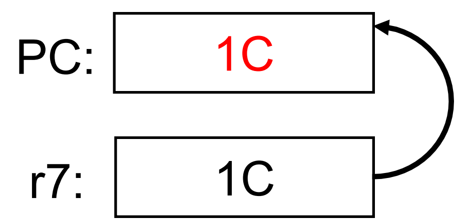
The contents of PC shown are those immediately after return was fetched

Subroutine

- LR: a dedicated register for br.sub instructions
  - br.sub: PC+2 is loaded into r7
  - return: PC is loaded with r7

```
            …
            …
0X1A:       BR.SUB
0X1C:       …

subroutine:
            …
0X88:       return
```



The contents of PC shown are those immediately after return completed execution

L-Format

- **Load/Store Instructions**

15 … 9	8	7 … 0
op-code	m.l	imm

Format L.1

15 … 9	8 … 6	5 … 3	2 … 0
op-code	r.dest	r.src	

Format L.2

e.g.: LOAD r1,@r2

or LOAD r.dest,r.src

LOAD r1,r2

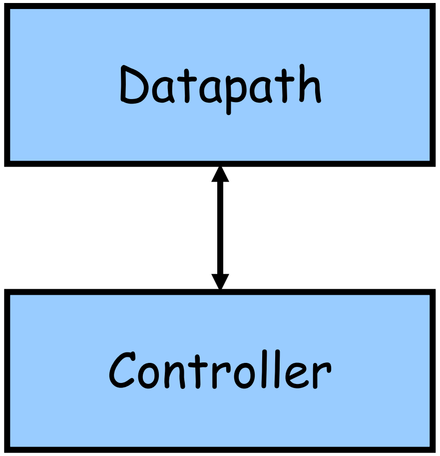
15 … 9	8 … 6	5 … 3	2 … 0
0010000	001	010	xxx

Project

- **A processor that executes every program written in the instruction set**

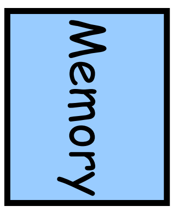
Processor Architecture

- 1)Datapath
  - Includes components, alu, register file, memory, …
- 2)Controller
  - Controls flow of instruction and data in datapath



Instruction Memory

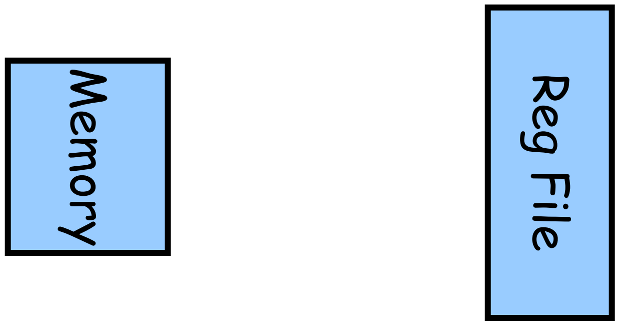
We need a container to hold instructions



Please see slide 4 and subsequent discussing memory specifications

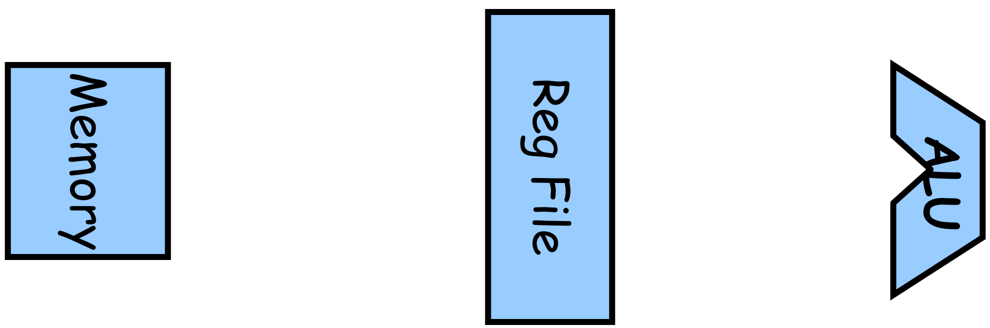
Register File

A place for r0…r7



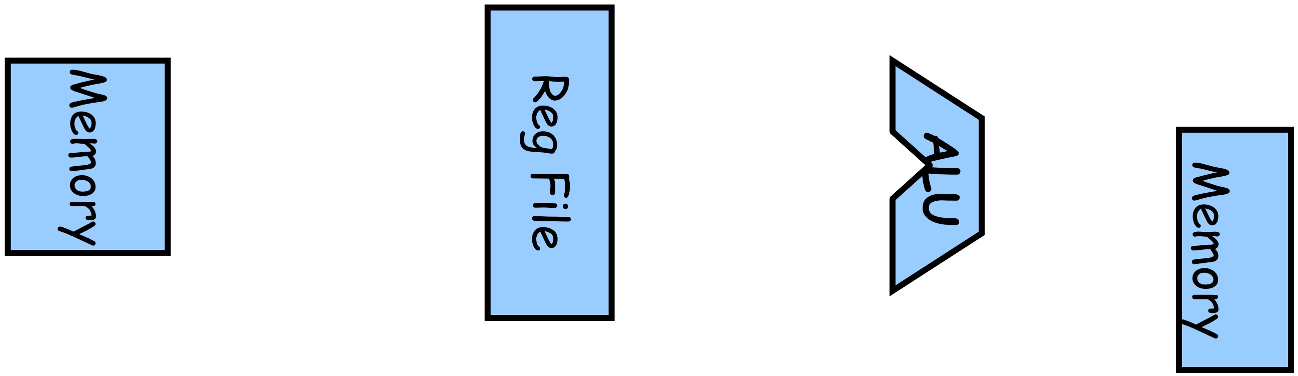
Alu

A unit for arithmetic calculations



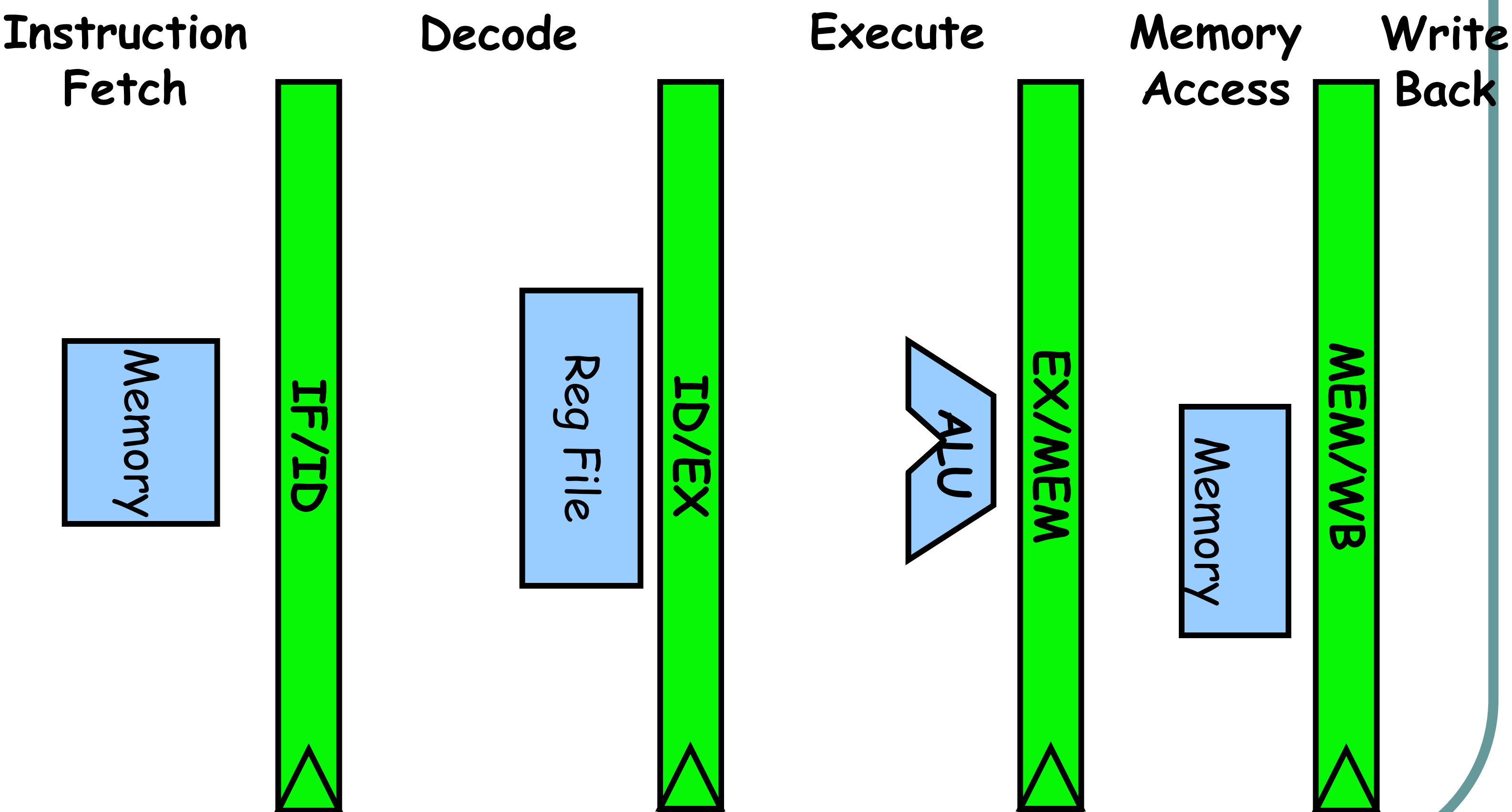
Data Memory

A unit that holds data



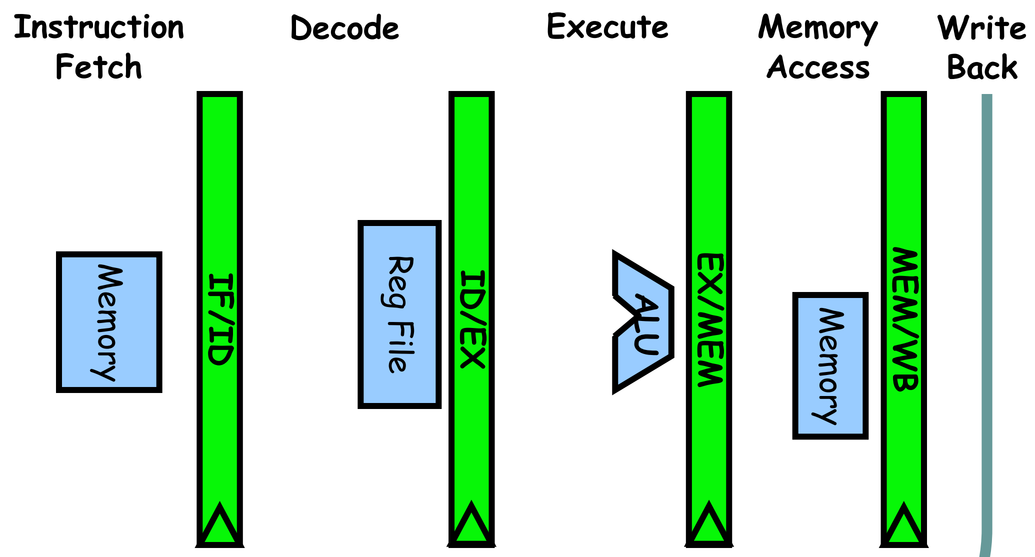
Pipeline Architecture

To break critical path



5-Stages Datapath

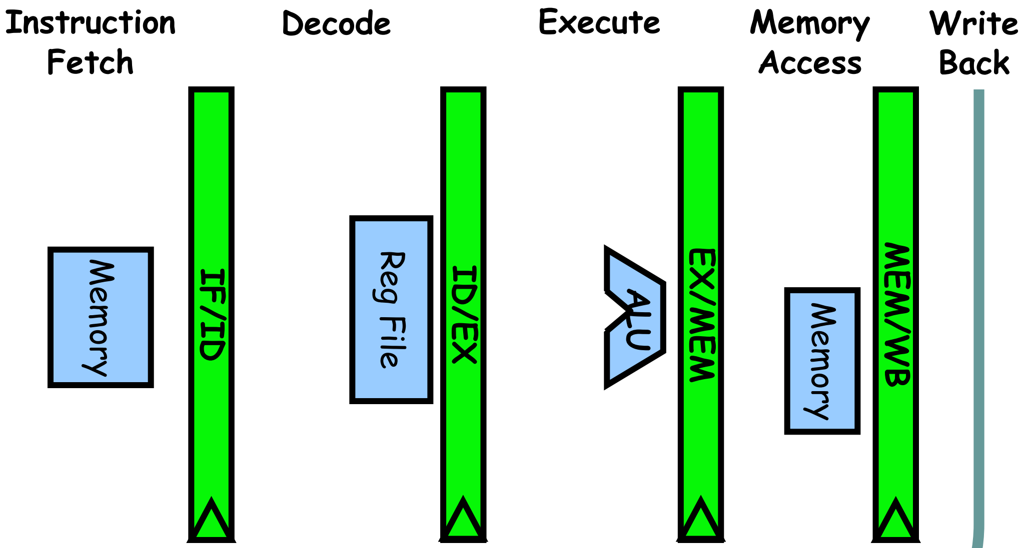
Up to now, design of main components



5-Stages Datapath

Up to now, design of main components

Complete the datapath for every instruction gradually

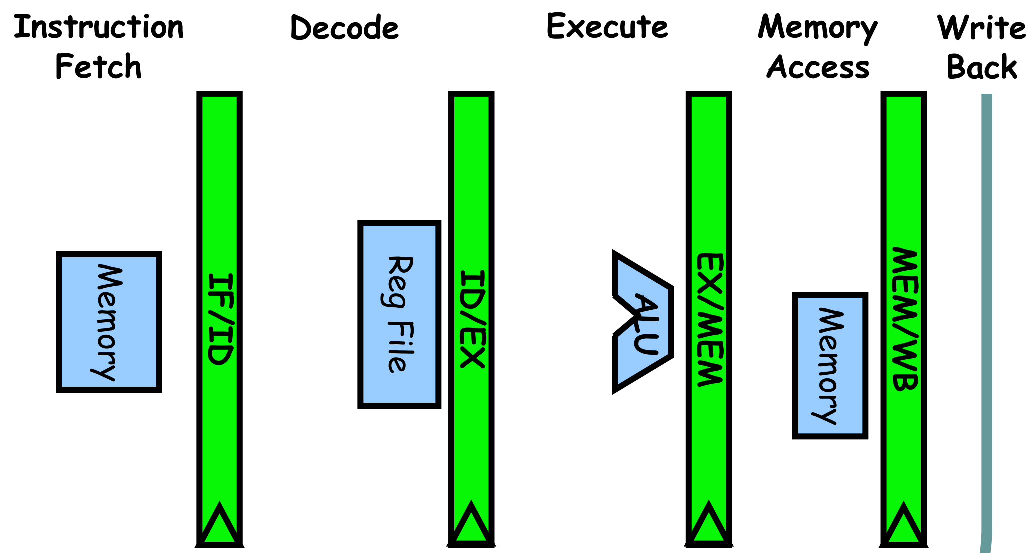


5-Stages Datapath

Up to now, design of main components

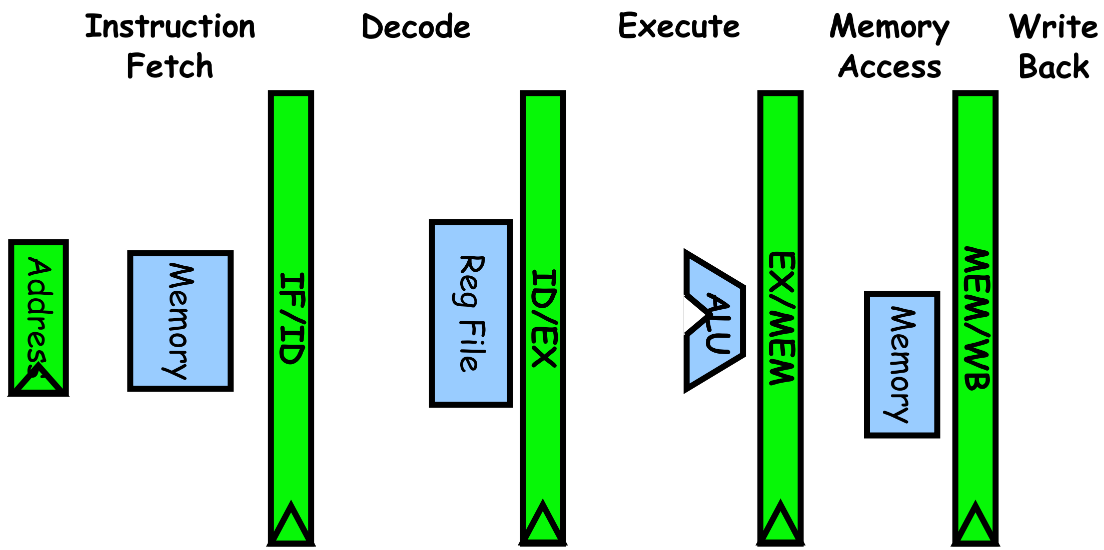
Complete the datapath for every instruction gradually

e.g. ADD instruction

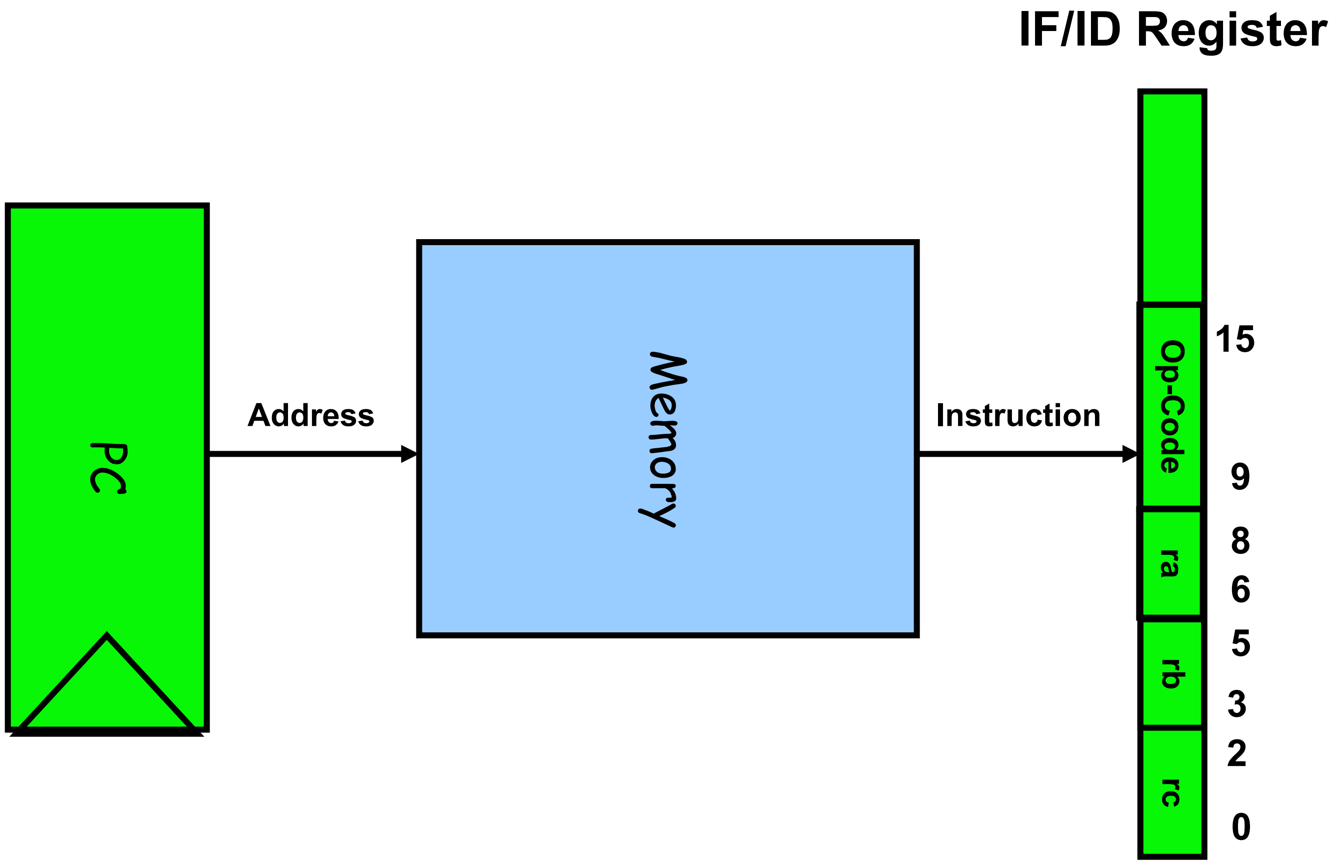


PC

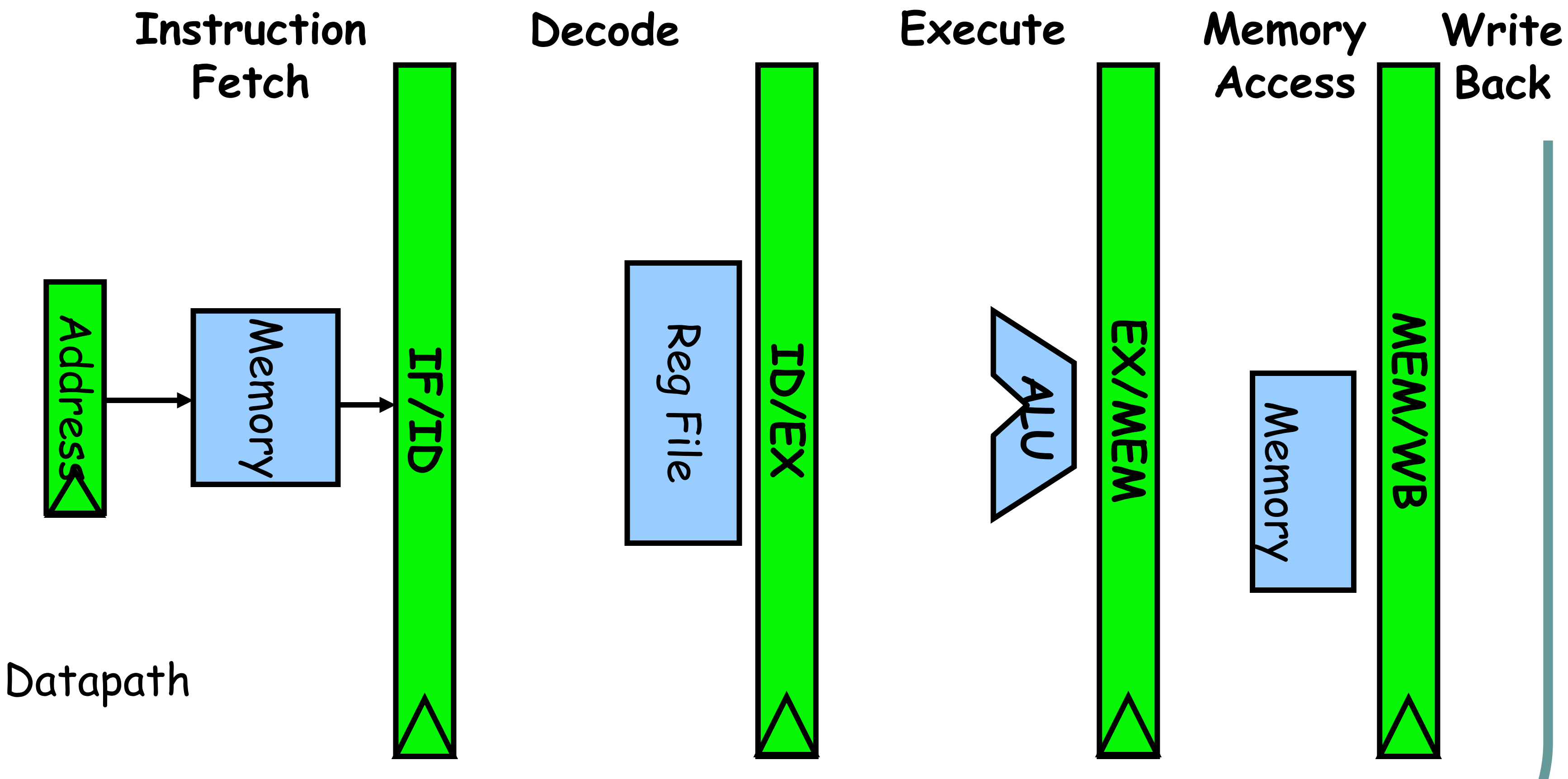
A component that holds address of Inst. Memory (PC)



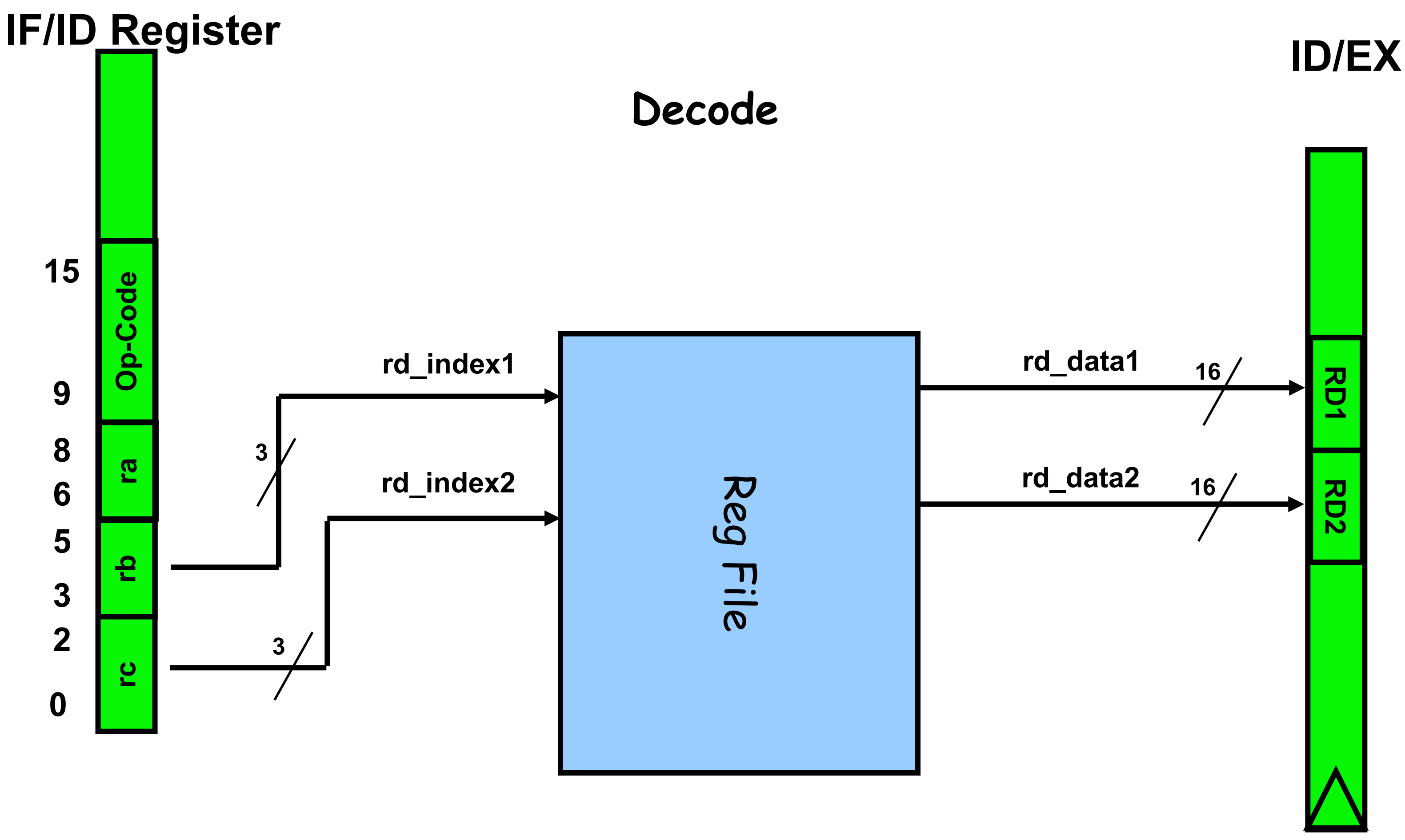
Fetch


IF/ID Register
PC
Address
Memory
Instruction
Op-Code
ra
rb
rc
15
9
8
6
5
3
2
0

ADD

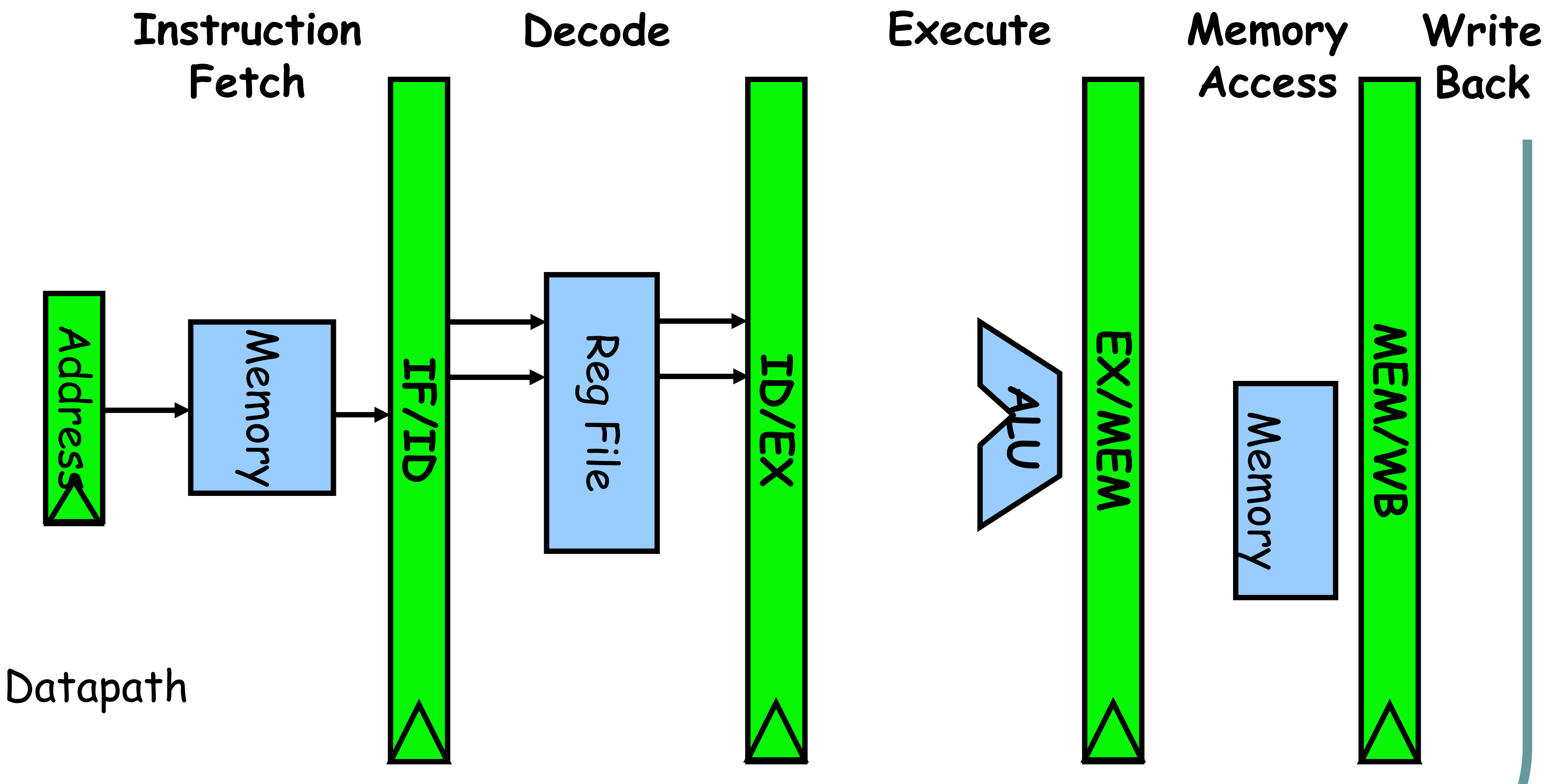

Instruction Fetch
Decode
Execute
Memory Access
Write Back
Address
Memory
IF/ID
Reg File
ID/EX
ALU
EX/MEM
Memory
MEM/WB
Datapath

Decode



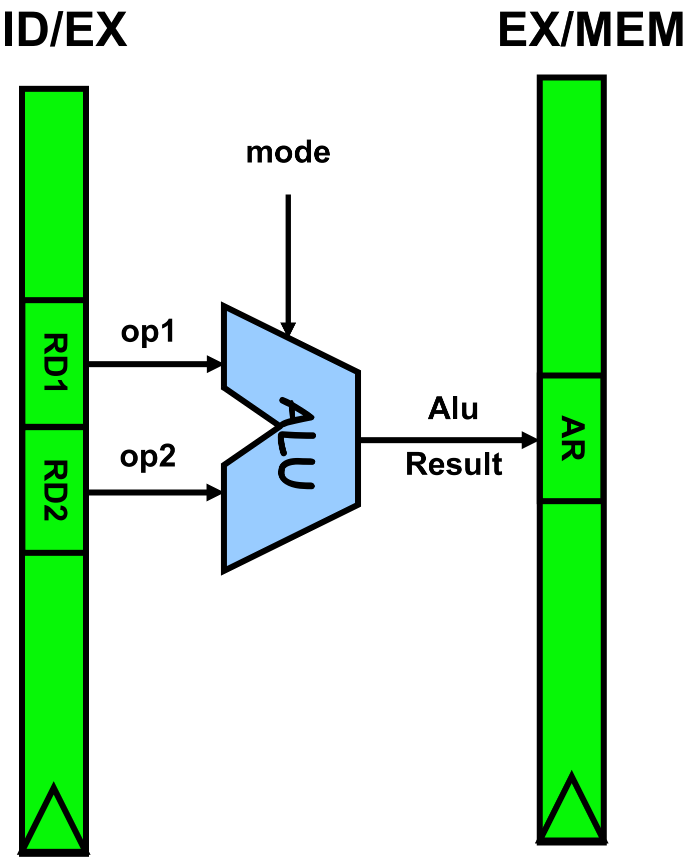
IF/ID Register
ID/EX
Decode
15
9
8
6
5
3
2
0
Op-Code
ra
rb
rc
3
3
rd_index1
rd_index2
Reg File
rd_data1
rd_data2
16
16
RD1
RD2

ADD

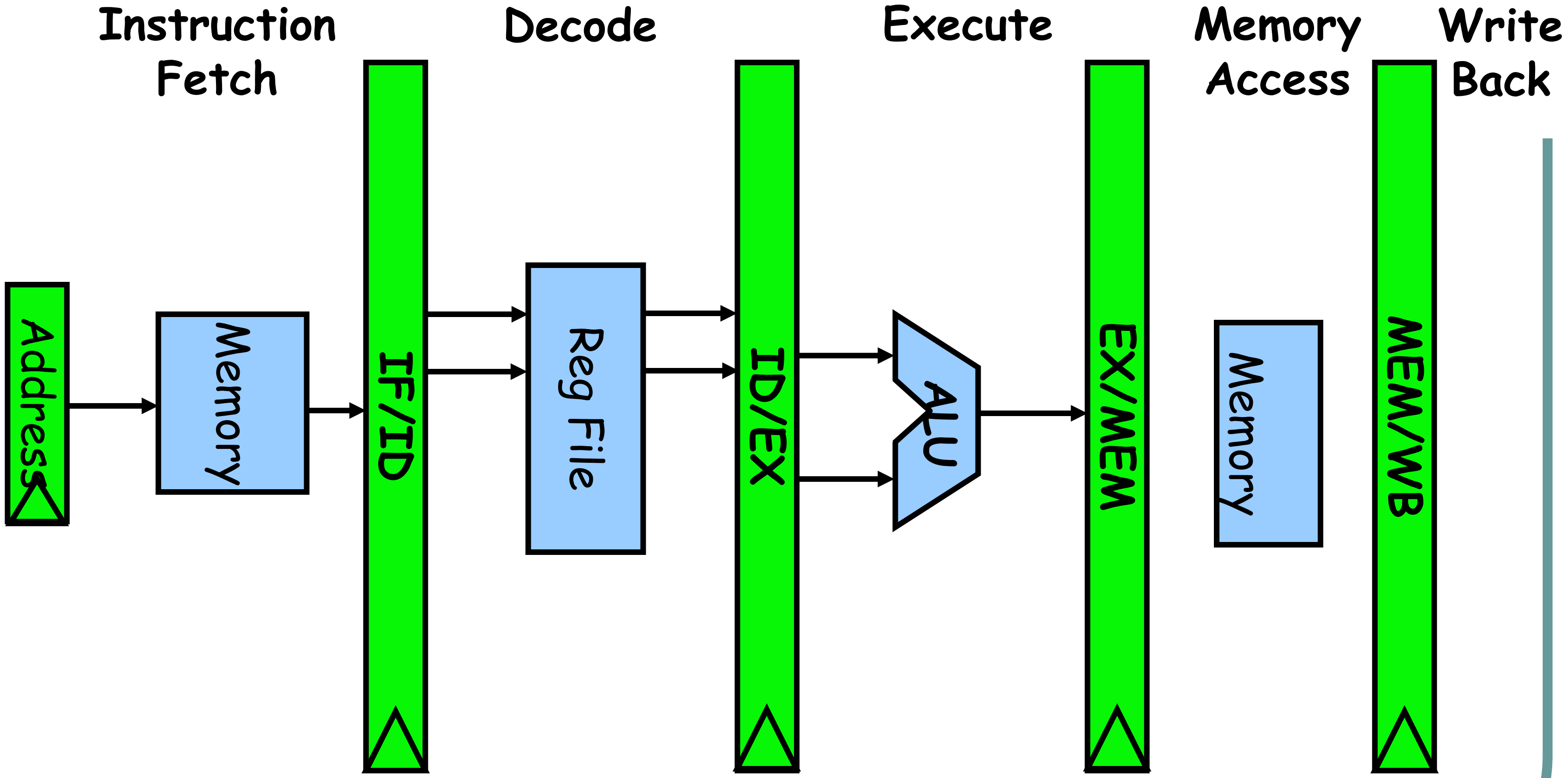


Instruction Fetch
Decode
Execute
Memory Access
Write Back
Address
Memory
IF/ID
Reg File
ID/EX
ALU
EX/MEM
Memory
MEM/WB
Datapath

Execution Stage

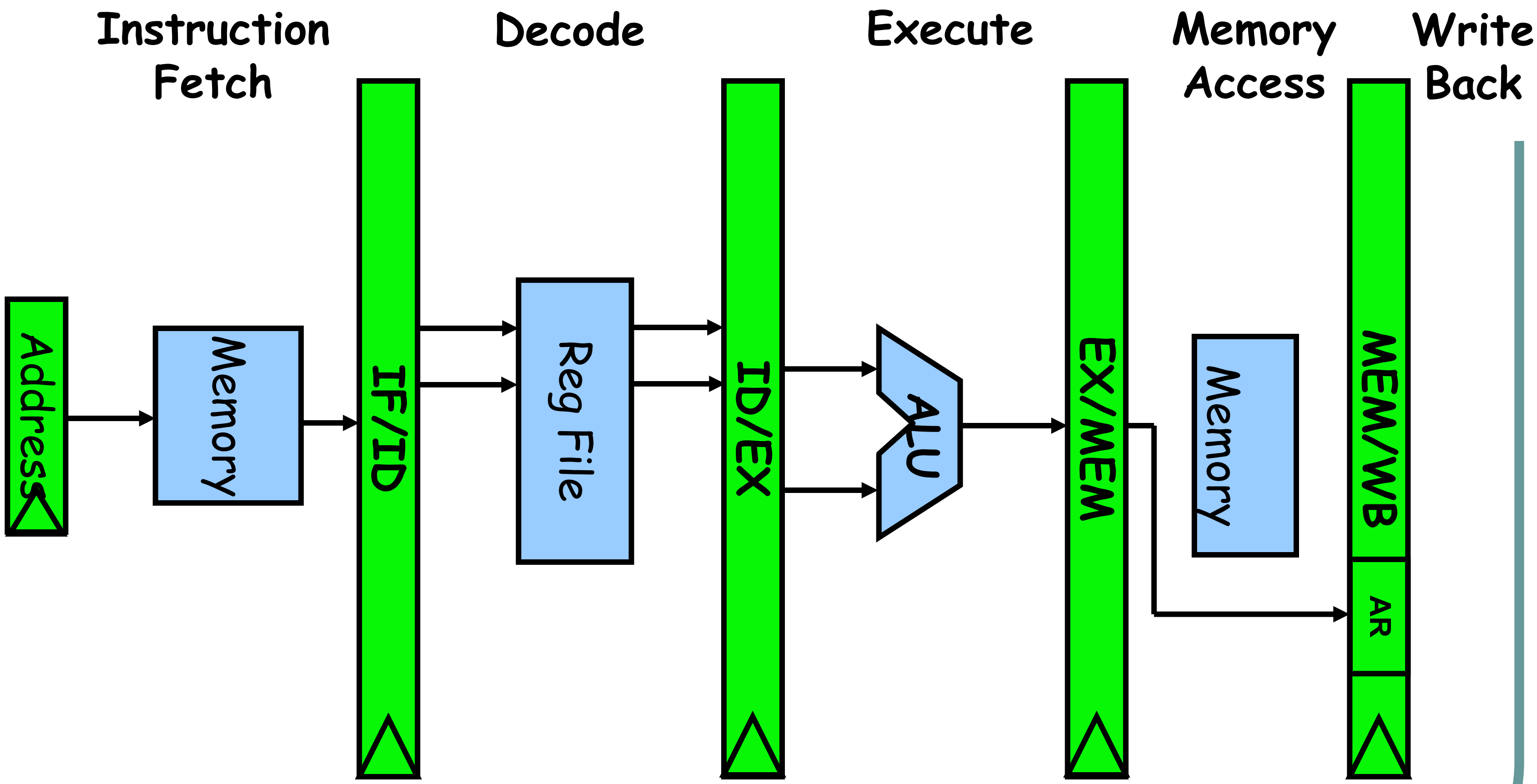


ADD

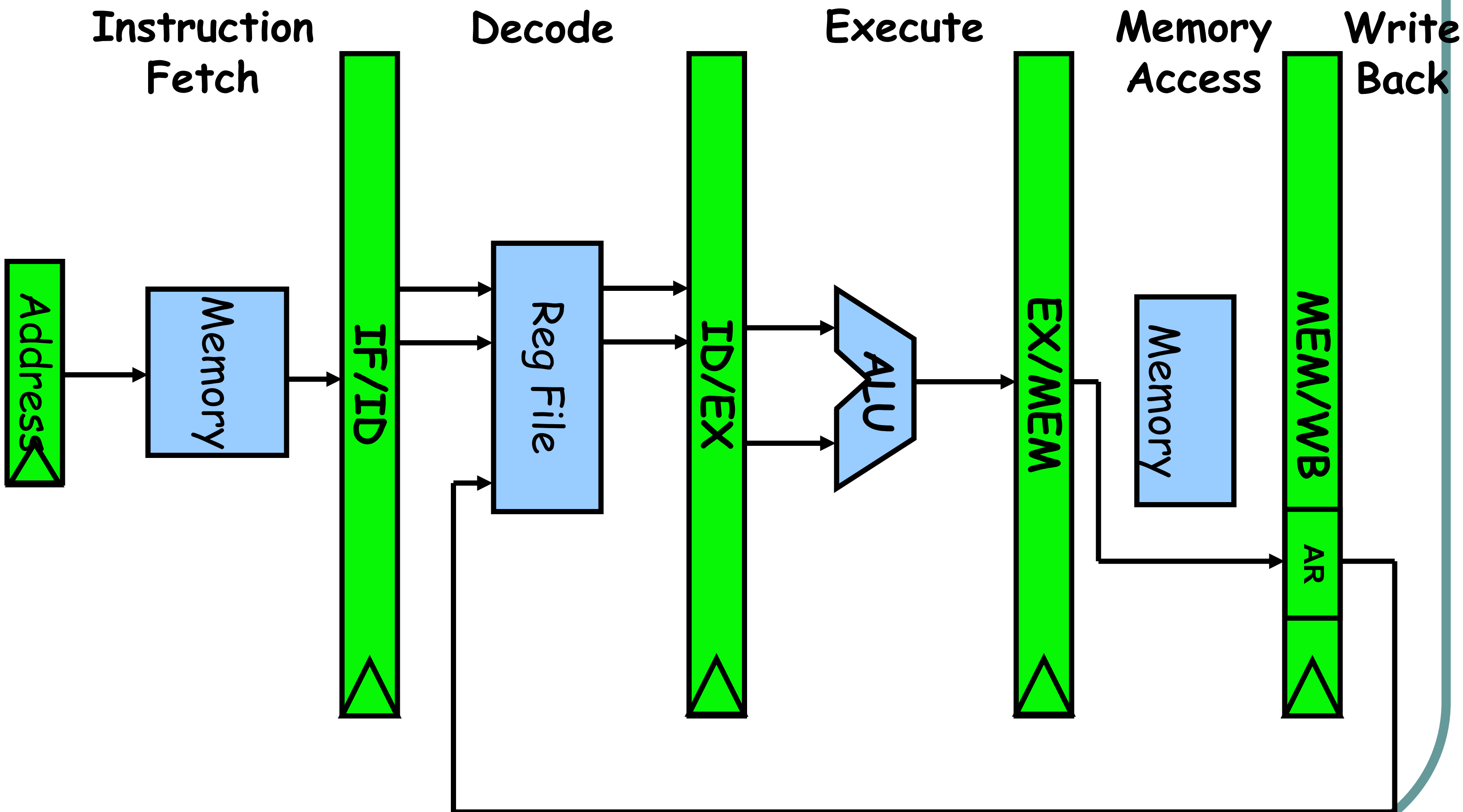


Instruction Fetch
Decode
Execute
Memory Access
Write Back
Address
Memory
IF/ID
Reg File
ID/EX
ALU
EX/MEM
Memory
MEM/WB

Memory Access

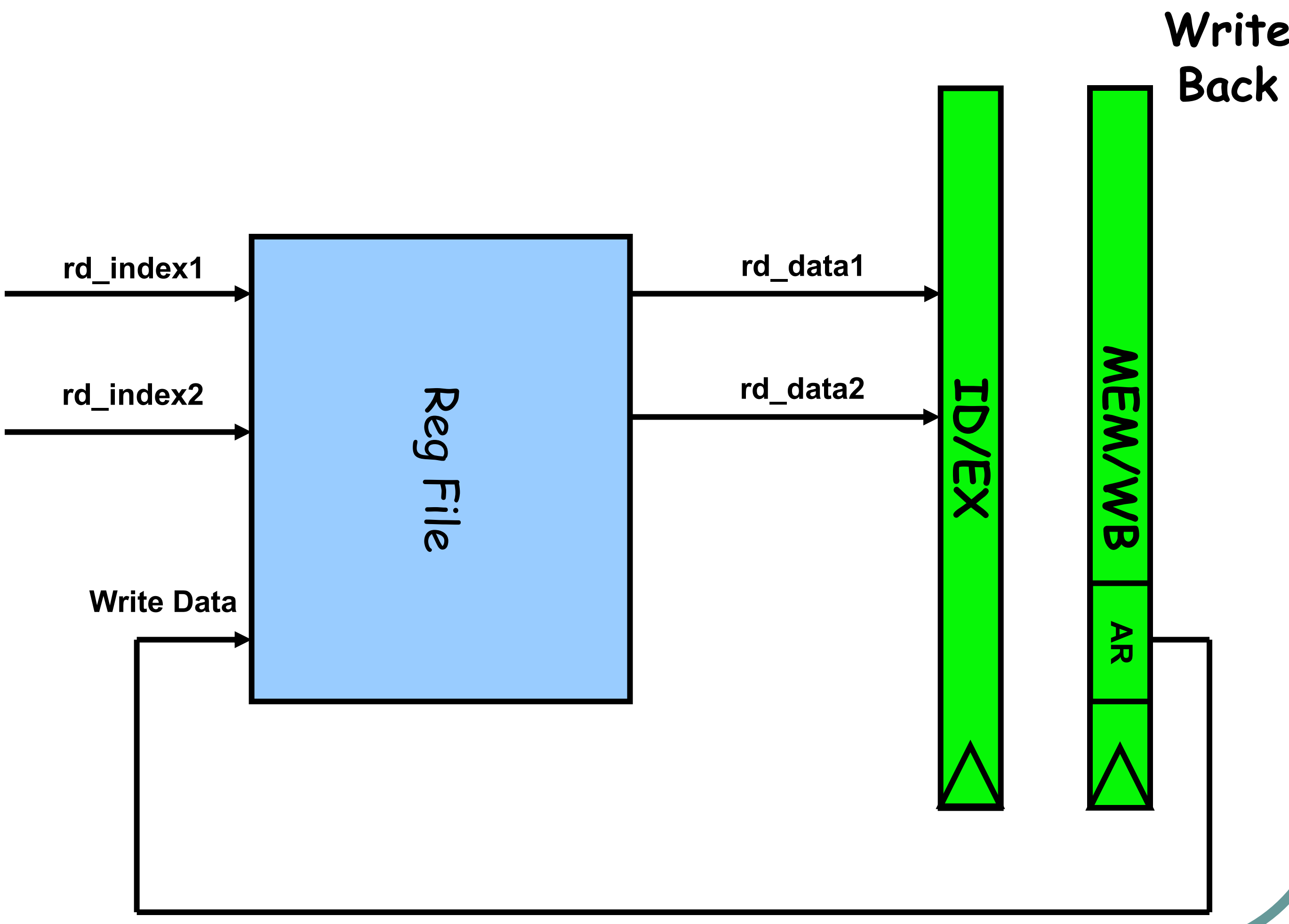


Write Back



Instruction Fetch
Decode
Execute
Memory Access
Write Back
Address
Memory
IF/ID
Reg File
ID/EX
ALU
EX/MEM
Memory
MEM/WB
AR

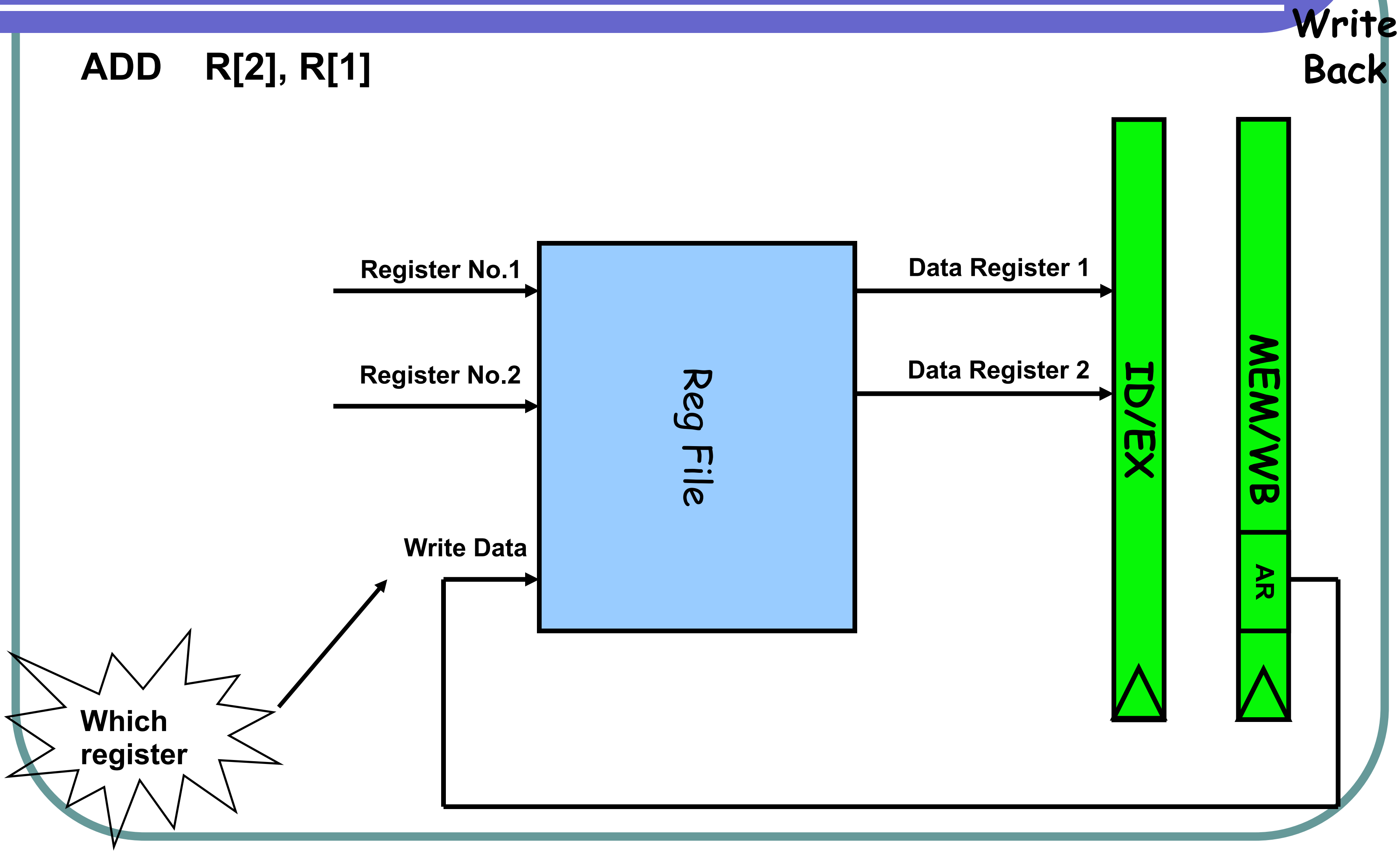
Write Back



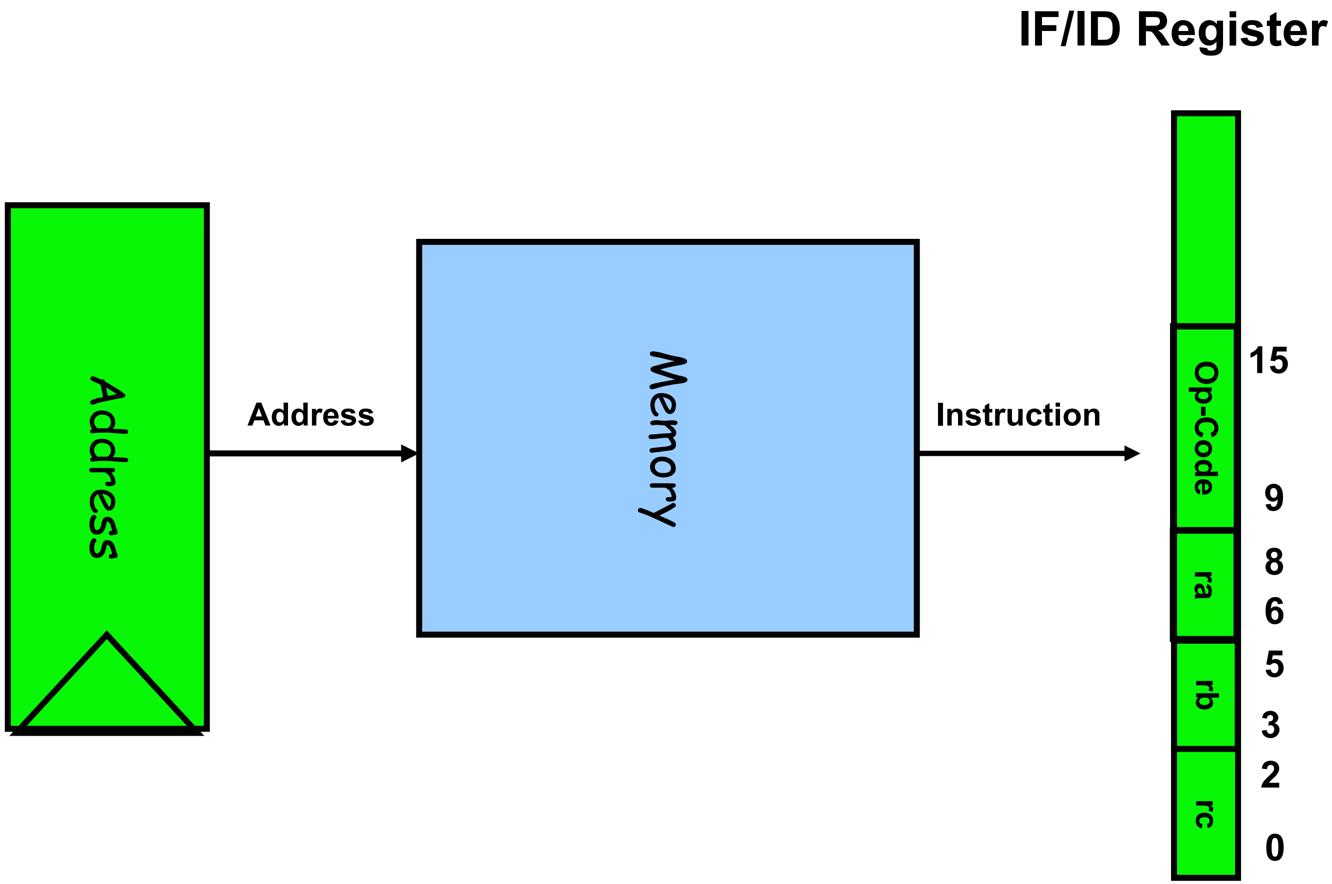
Write Back
rd_index1
rd_index2
Write Data
Reg File
rd_data1
rd_data2
ID/EX
MEM/WB
AR

Write Back

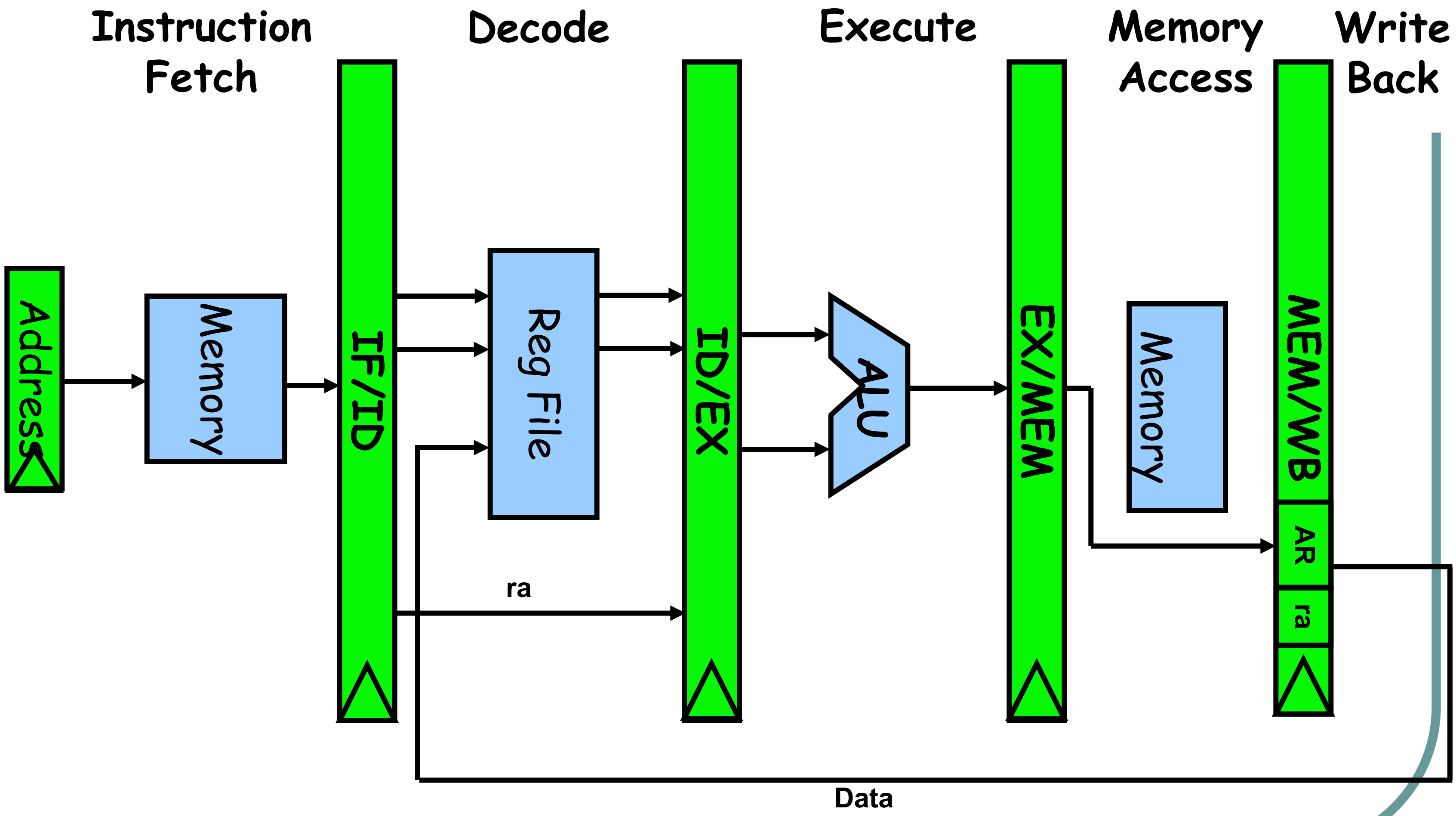
ADD R[2], R[1]



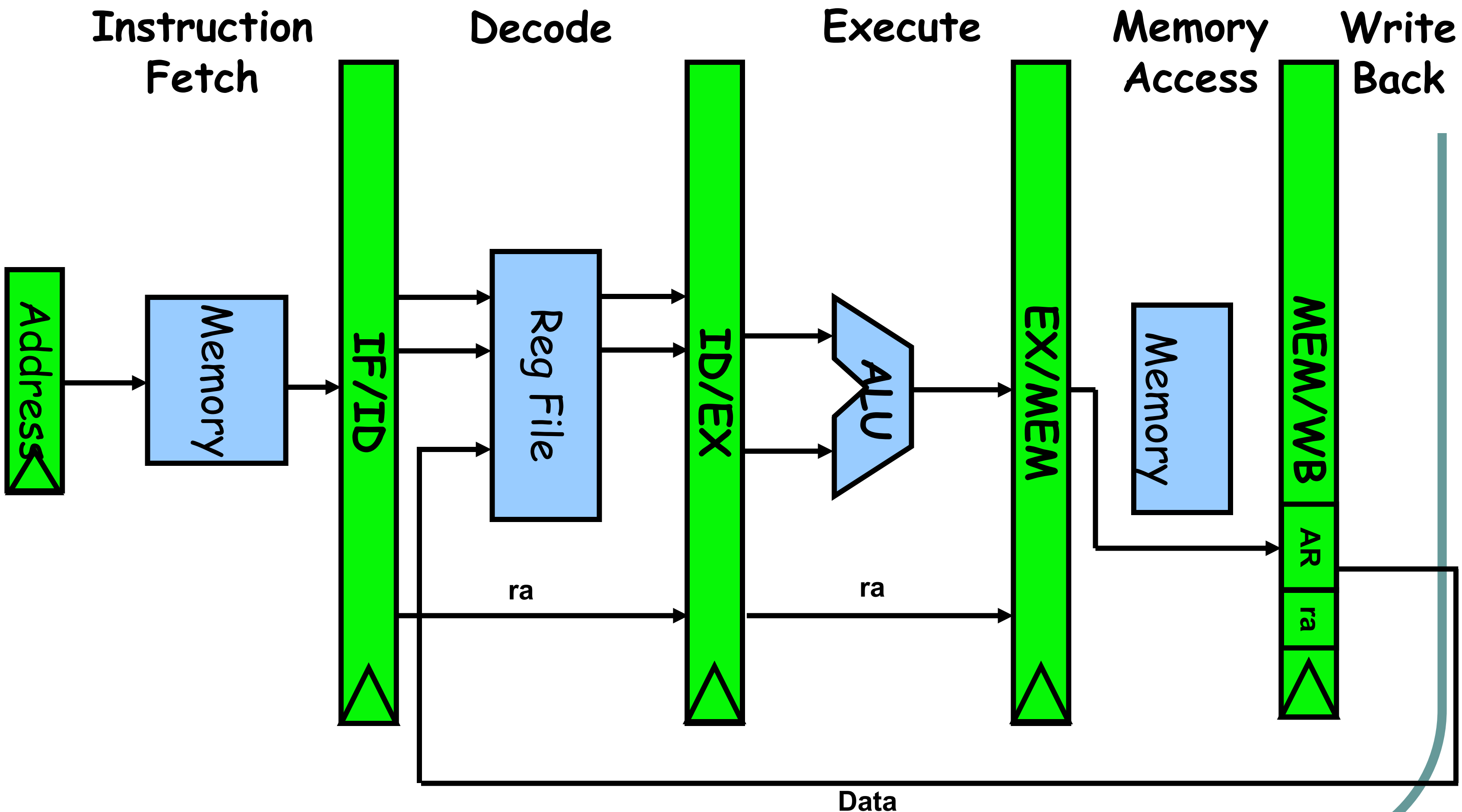
Revising Our Design



Write Back

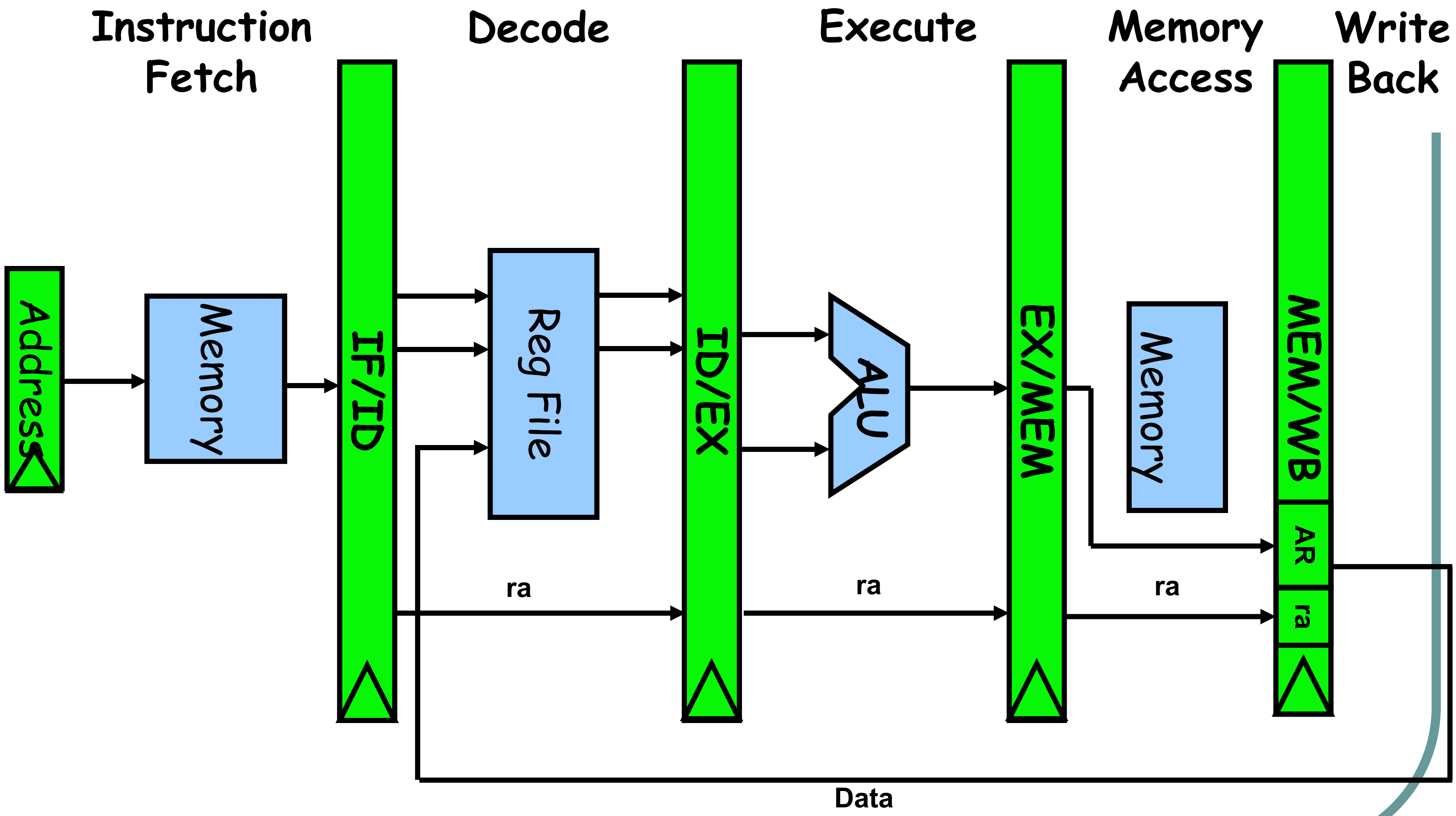

Instruction Fetch
Decode
Execute
Memory Access
Write Back
Address
Memory
IF/ID
Reg File
ID/EX
ALU
EX/MEM
Memory
MEM/WB
AR
ra
ra
Data

Write Back

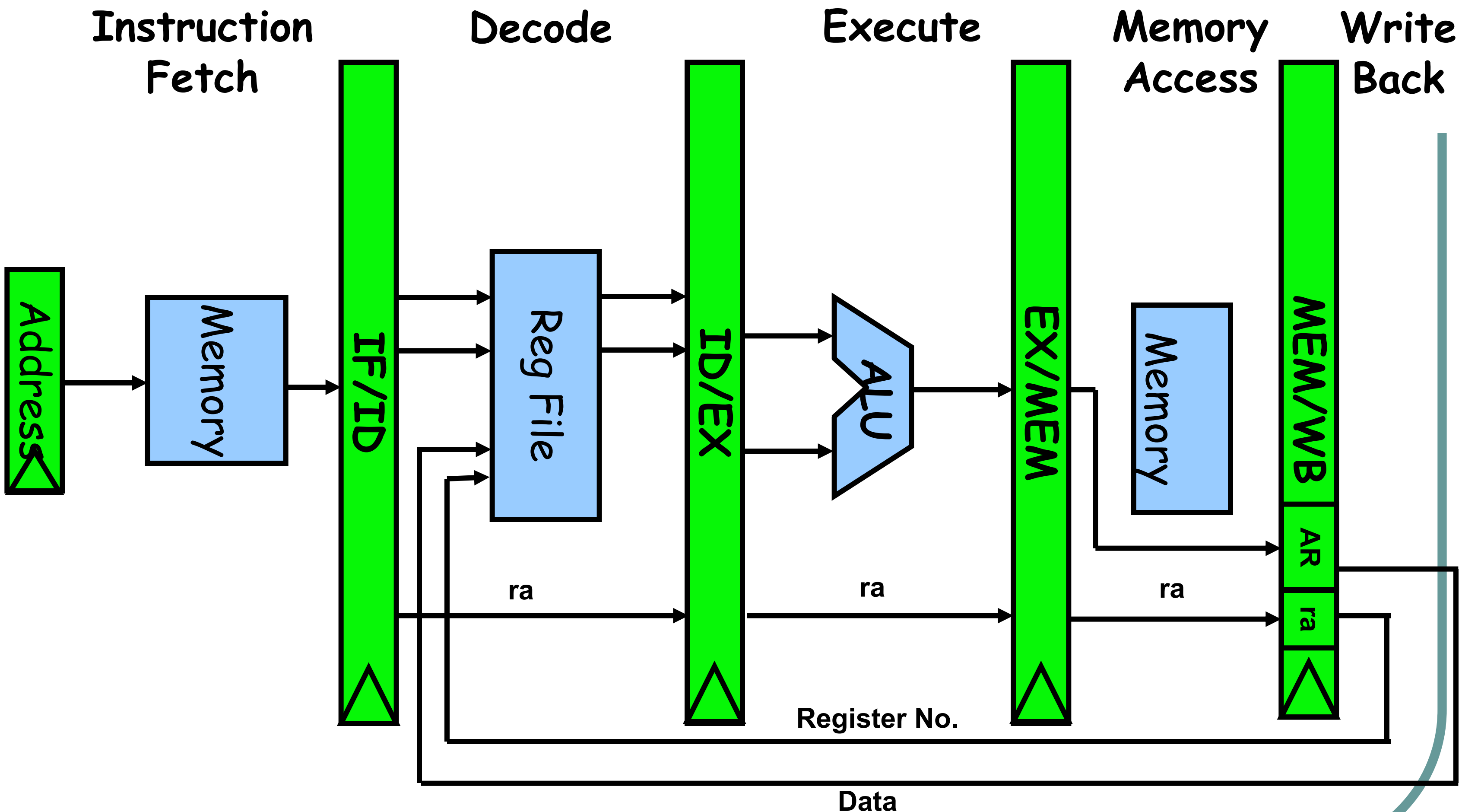


Instruction Fetch
Decode
Execute
Memory Access
Write Back
Address
Memory
IF/ID
Reg File
ID/EX
ALU
EX/MEM
Memory
MEM/WB
AR
ra
ra
ra
Data

Write Back


Instruction Fetch
Decode
Execute
Memory Access
Write Back
Address
Memory
IF/ID
Reg File
ID/EX
ALU
EX/MEM
Memory
MEM/WB
AR
ra
ra
ra
ra
Data

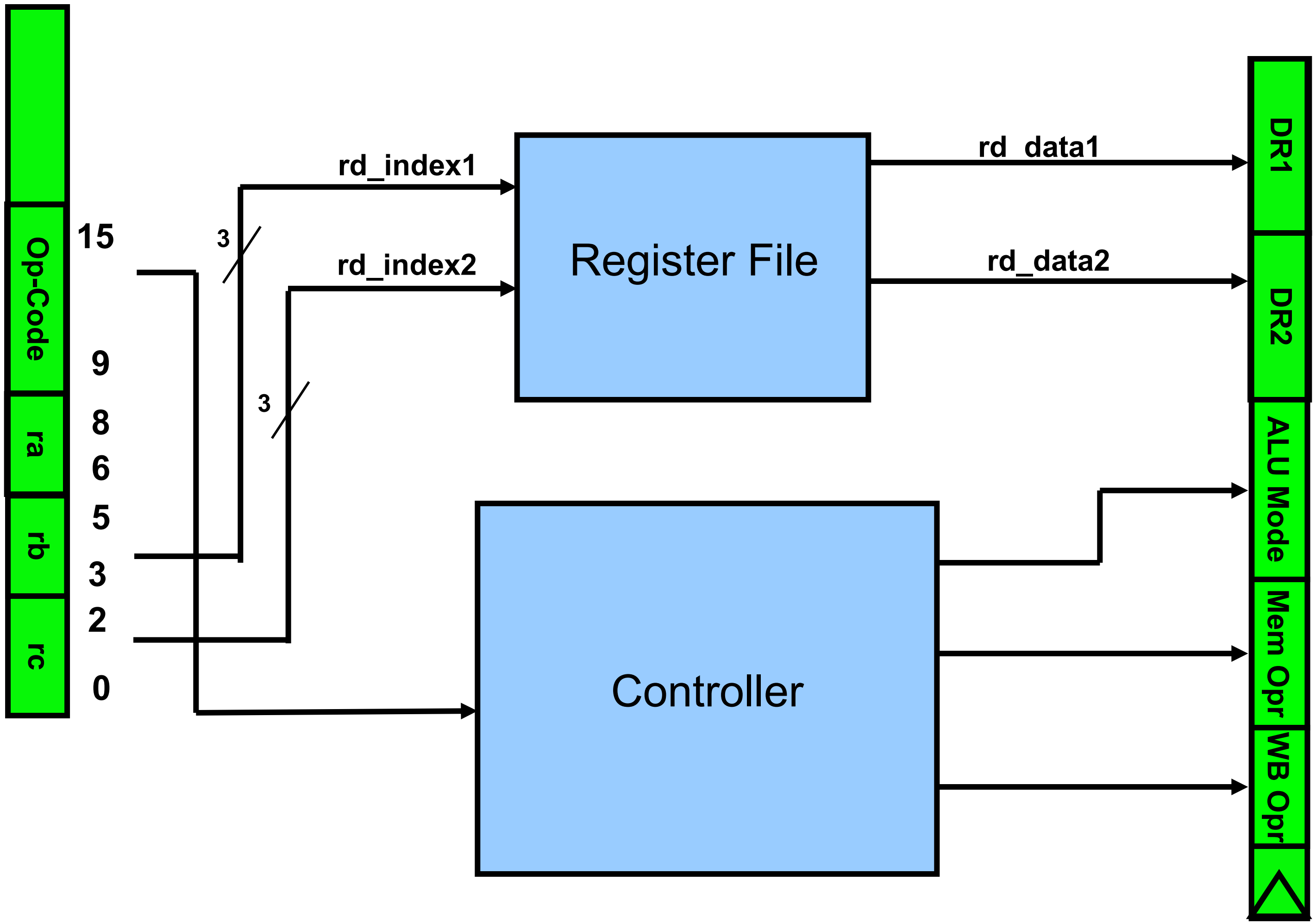
Write Back


Instruction Fetch
Decode
Execute
Memory Access
Write Back
Address
Memory
IF/ID
Reg File
ID/EX
ALU
EX/MEM
Memory
MEM/WB
AR
ra
ra
ra
ra
Register No.
Data

Other Instructions

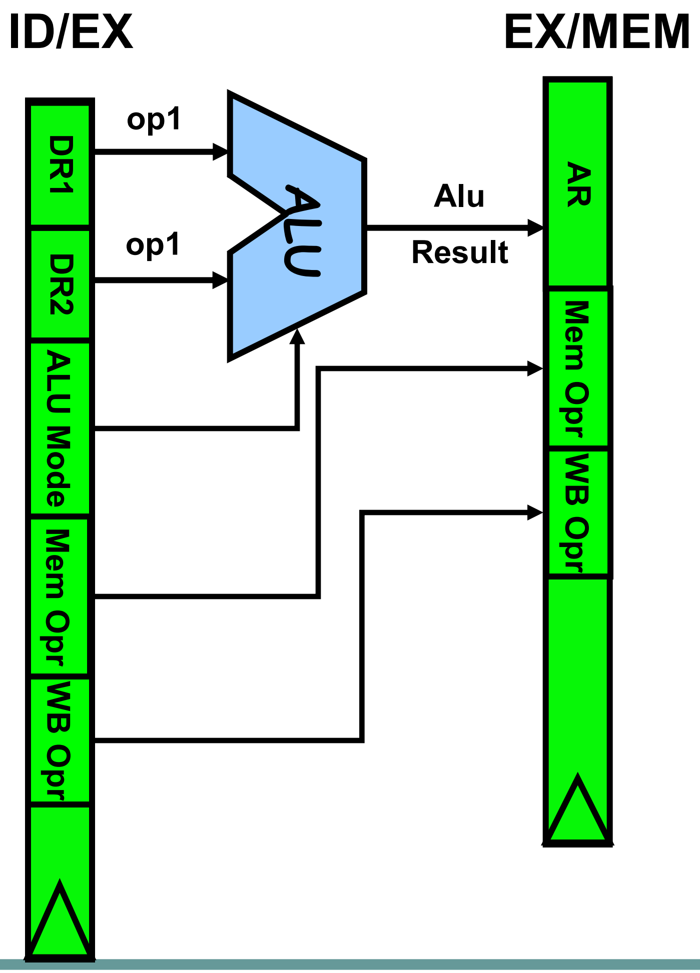
- We should repeat similar steps for other instructions

Controller



The controller controls all the stages of the pipeline.
It may be distributed, hierarchical or centralized

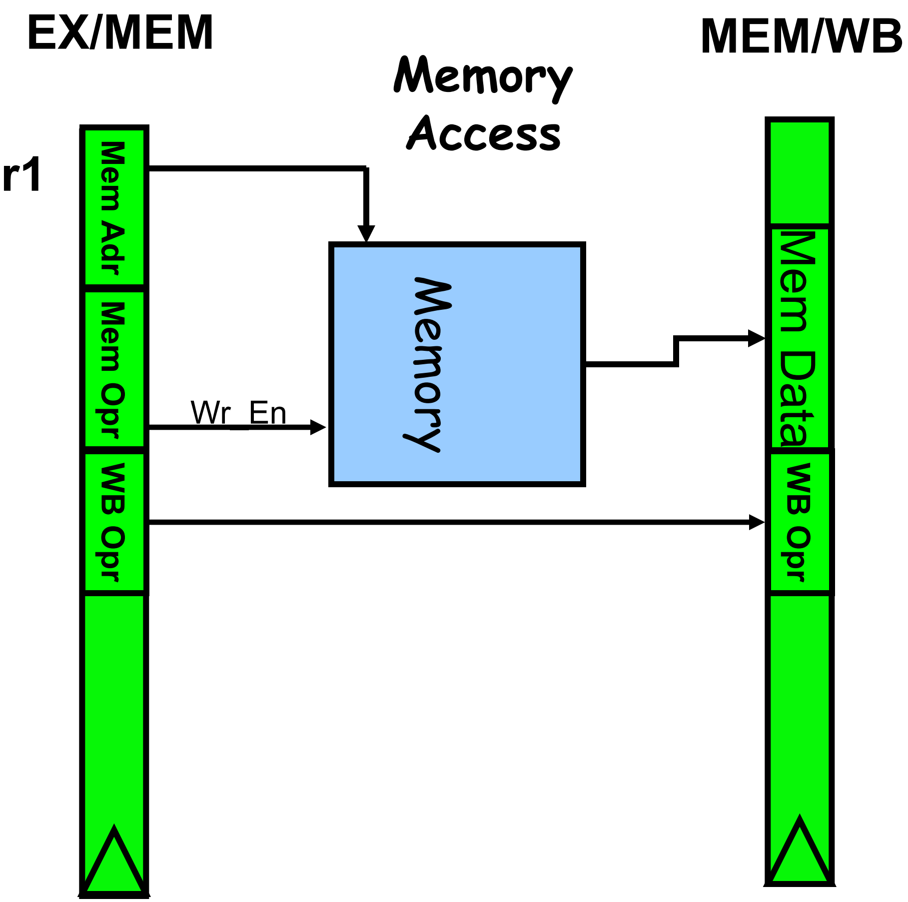
Execute



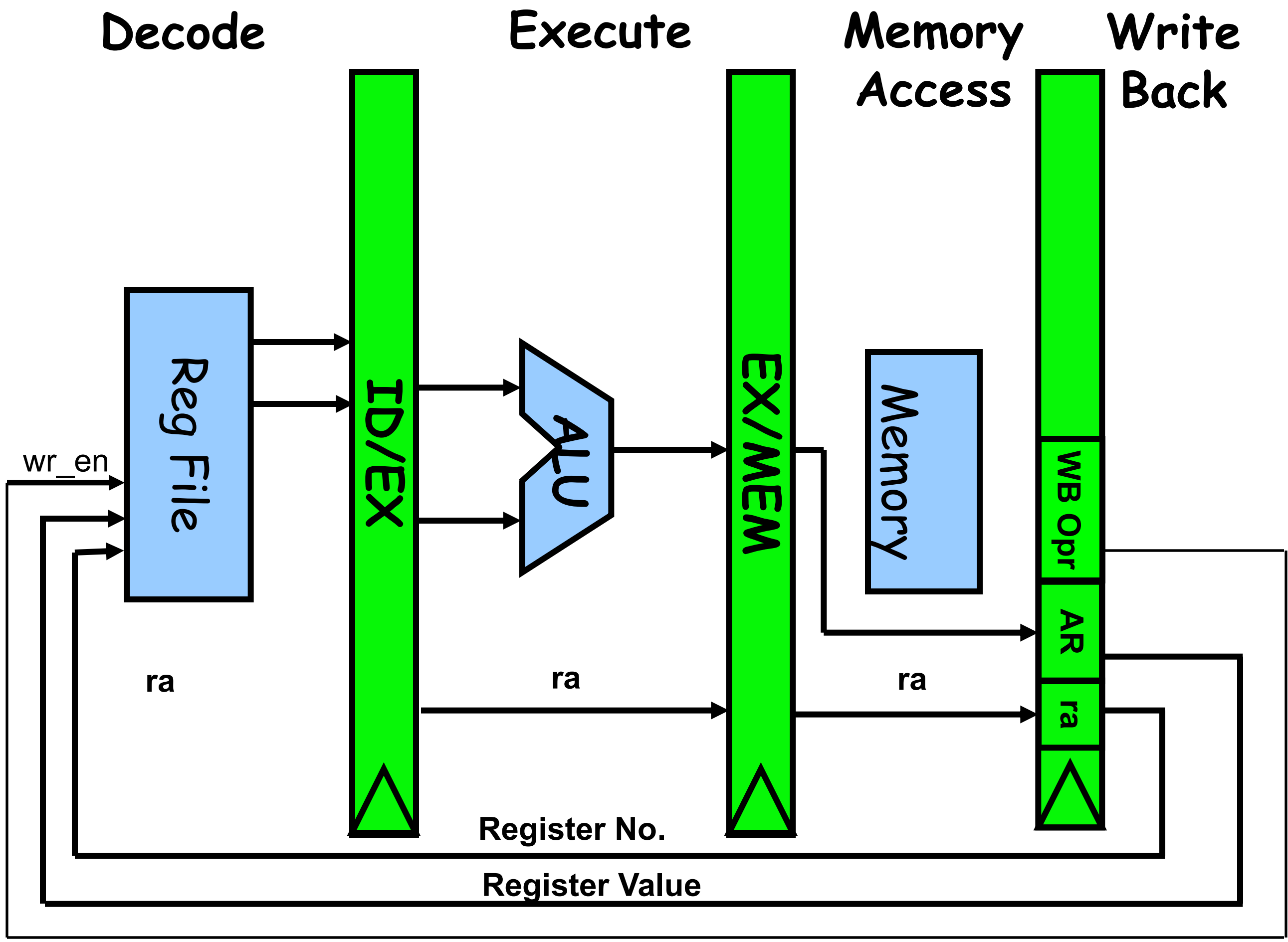
ID/EX
EX/MEM
DR1
DR2
ALU Mode
Mem Opr
WB Opr
op1
op1
ALU
Alu
Result
AR
Mem Opr
WB Opr

Memory Access

STORE @r2, r1
Mem Opr:
Write_En = 0



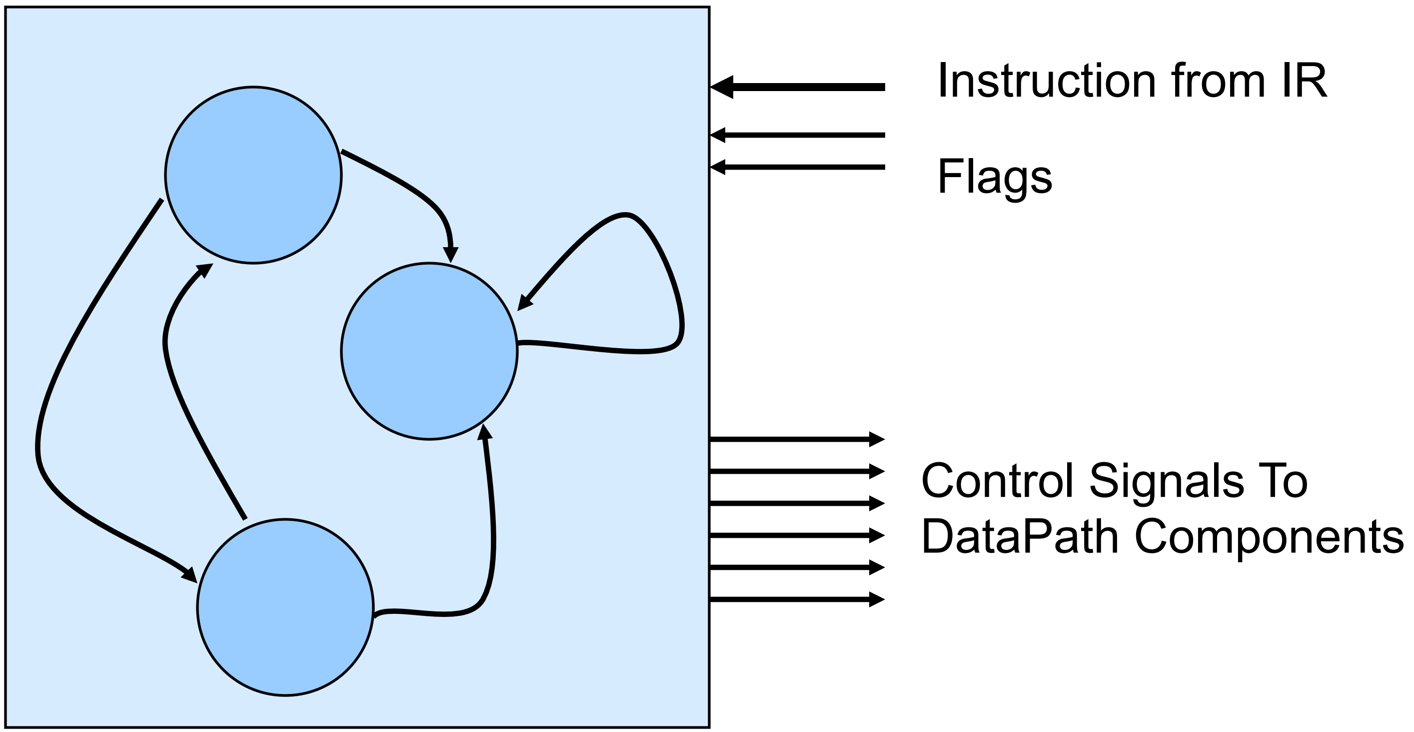
Write Back



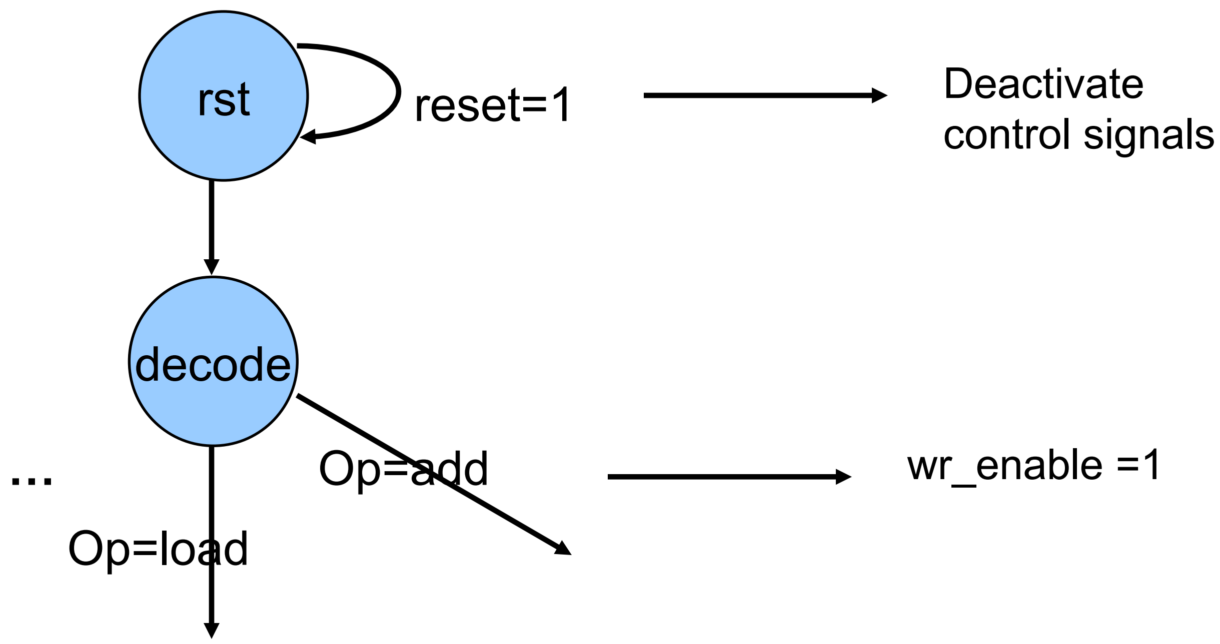
Decode
Execute
Memory Access
Write Back
Reg File
ID/EX
ALU
EX/MEM
Memory
WB Opr
AR
ra
wr_en
ra
ra
ra
Register No.
Register Value

Controller

Design Controller Base on Control Signals in Data-Path



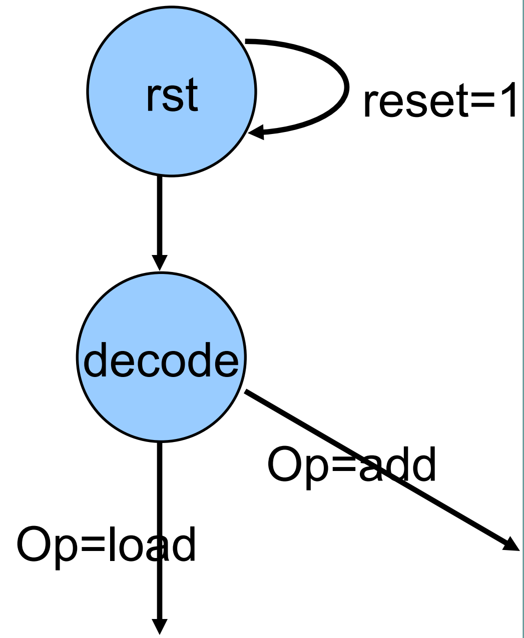
State Machine



State Machine Implementation

```
parameter [3:0]
        RESET=0,DECODE=1;

always @(negedge clk)
        if(rst)
                begin
                state = RESET;
                //deactivate all control signals
                end
        else
                begin
                case(state)
                        RESET:  begin
                                state=DECODE;
                        end
                        DECODE: begin
                                if(opcode=ADD)
                                …
                        end
                        default:
                                state=RESET;
                endcase
                end
```



The reset state needs to be modified to account for the two different resets as per slides 2 & 3

Implementation Strategy

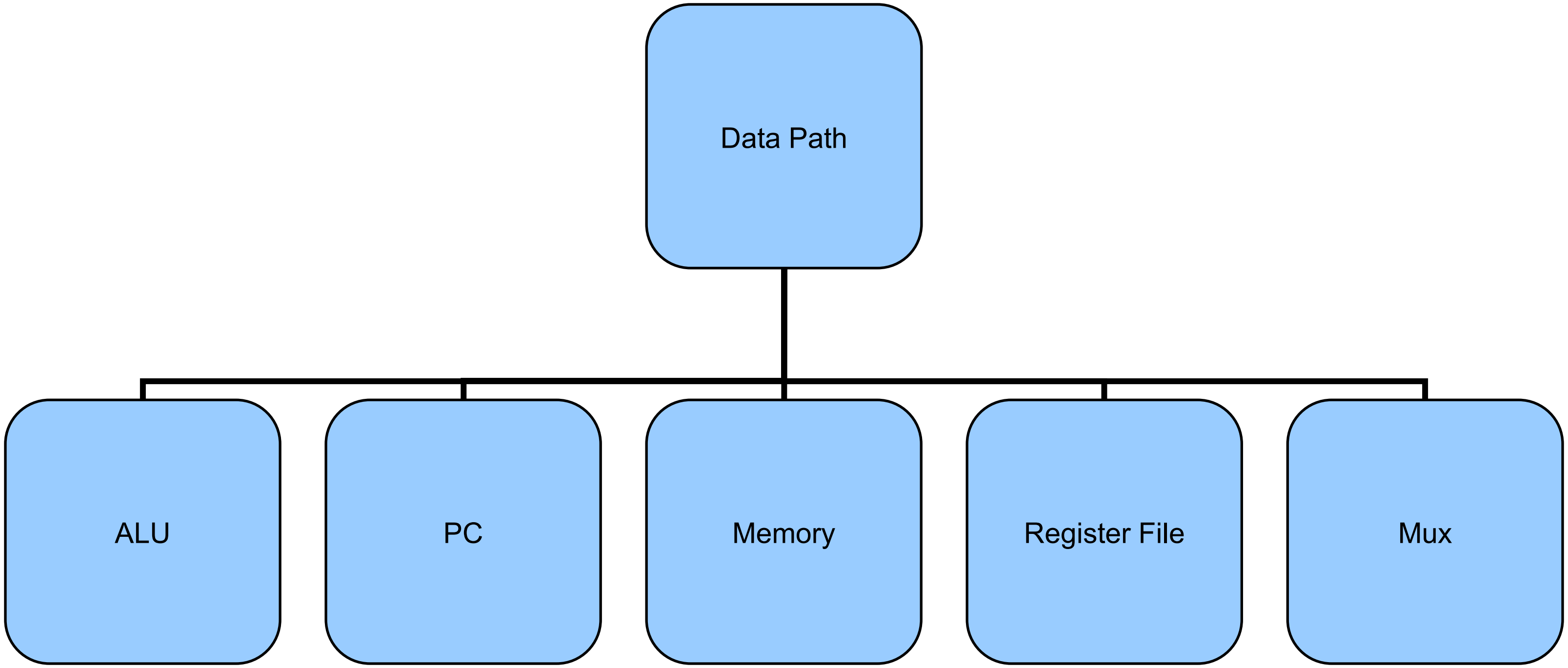
- First Design Datapath
- Design Controller base on Data-Path
- Connect Controller and Data-Path

Designing Datapath

- Necessary Components
  - Program Counter
  - Instruction/Data Memory
  - ALU
  - Register File
  - ….

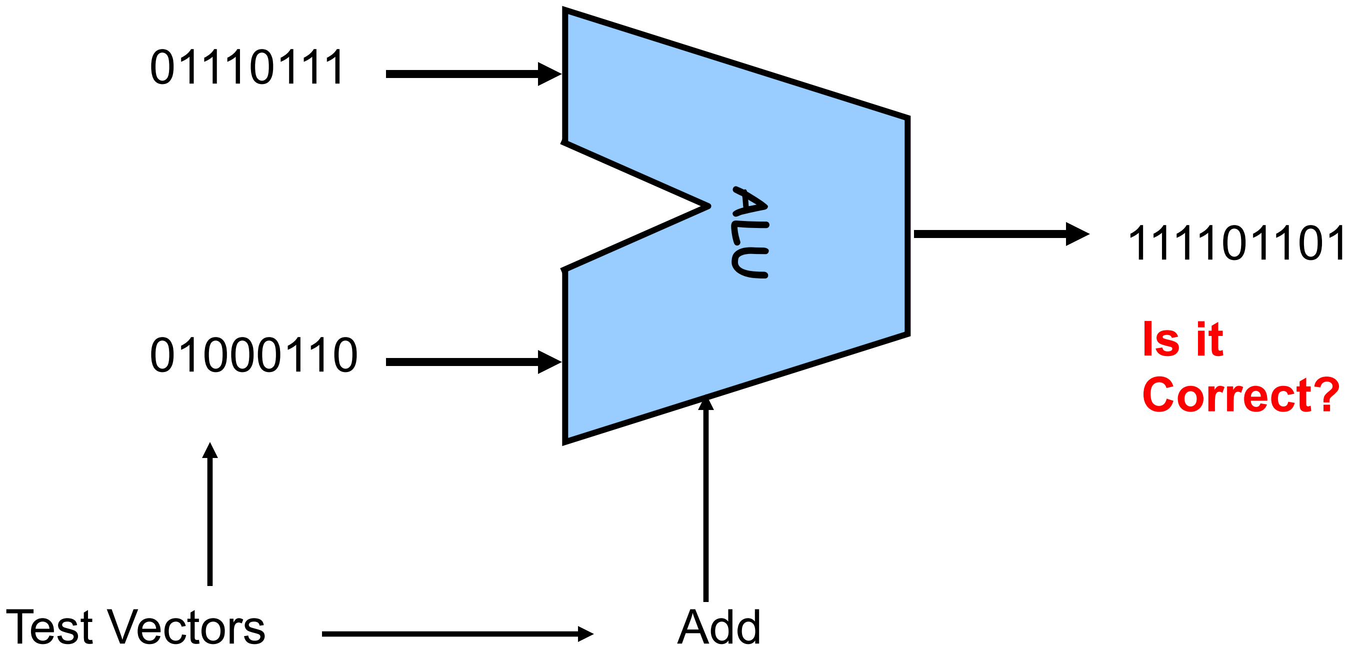
Datapath Hierarchy

Start your design from Bottom Modules

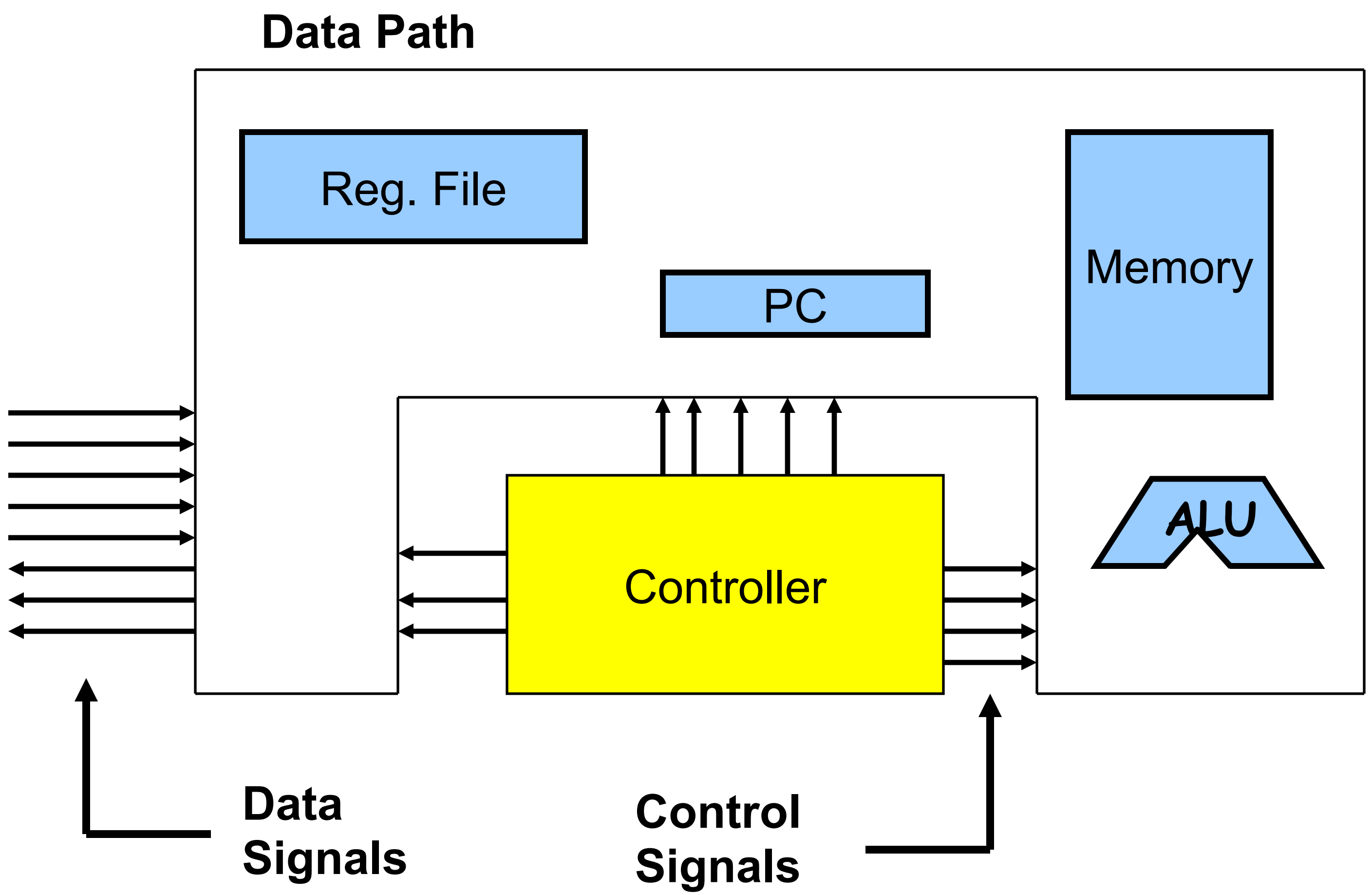


Simulate each Component

- Post-Route Simulation



Wiring CPU (Top Module)



Designing for Hazards

- Hazards Because of Pipeline
  - Data Hazard
  - Control Hazard

Simulate and Implement

- Simulating Complete CPU
- Implementation (Pin Assignment)