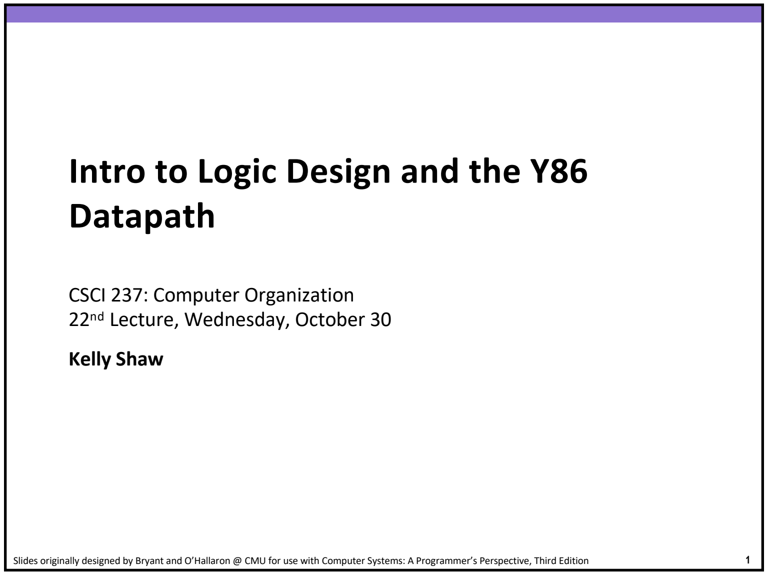
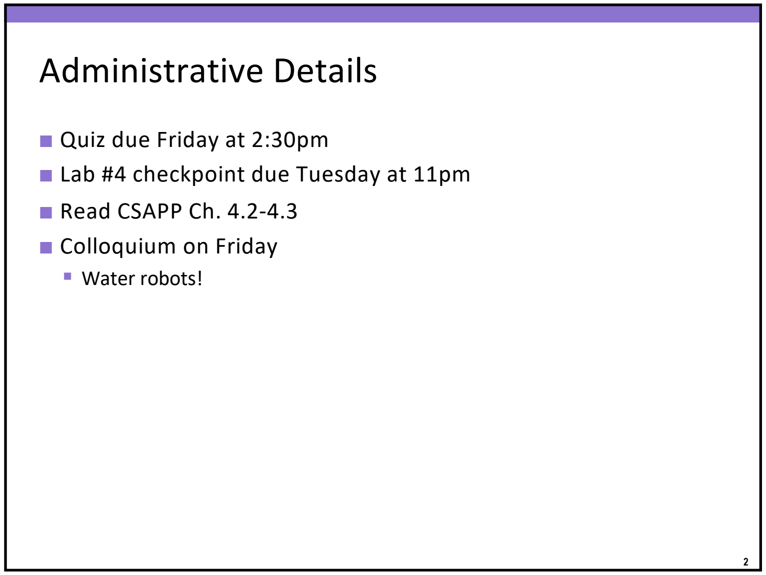


Intro to Logic Design and the Y86 Datapath

CSCI 237: Computer Organization
22nd Lecture, Wednesday, October 30

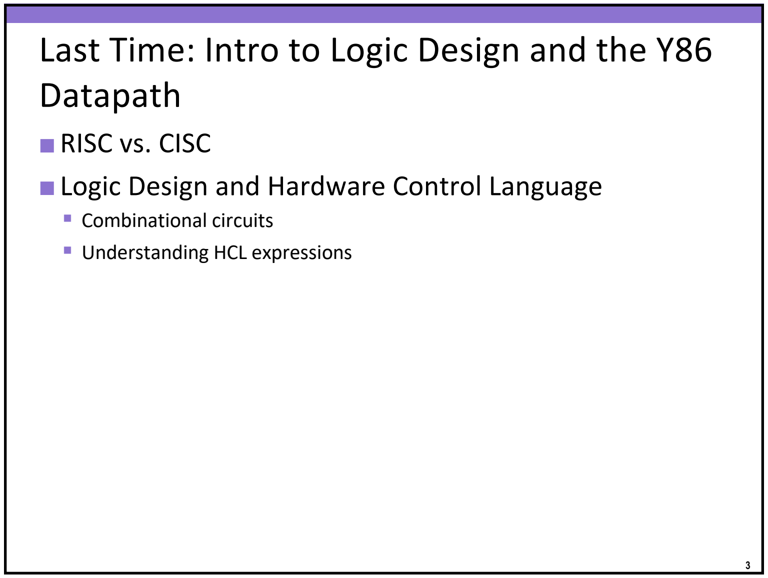
Kelly Shaw

Slides originally designed by Bryant and O'Hallaron @ CMU for use with Computer Systems: A Programmer's Perspective, Third Edition



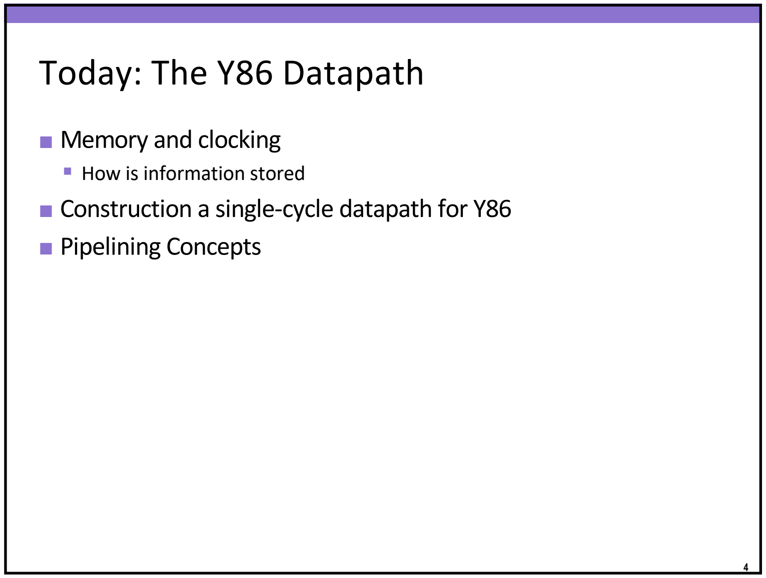
Administrative Details

- Quiz due Friday at 2:30pm
- Lab #4 checkpoint due Tuesday at 11pm
- Read CSAPP Ch. 4.2-4.3
- Colloquium on Friday
  - Water robots!



Last Time: Intro to Logic Design and the Y86 Datapath

- RISC vs. CISC
- Logic Design and Hardware Control Language
  - Combinational circuits
  - Understanding HCL expressions



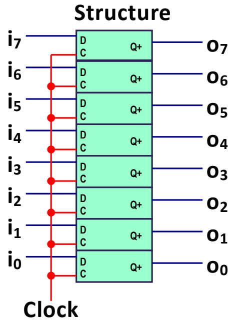
Today: The Y86 Datapath

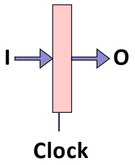
- Memory and clocking
  - How is information stored
- Construction a single-cycle datapath for Y86
- Pipelining Concepts

Moving on: Storing Bits

- Combinational circuits do not store information
  - Only react to signals at inputs and generate outputs
- Creating a *sequential circuit* requires storage
  - Seq circuits have state and perform computations on that state
- Storage devices are controlled by a single *clock*
  - A periodic signal that determines when new values are to be loaded
- Two classes of memory devices:
  - Clocked registers – store individual bits or words
  - Random access memories – store multiple words using an address to select where word should be read/written
- Distinction between hardware registers and program registers
  - Hardware registers are directly connected to circuits
  - Program registers are stored in register file, which is a type of RAM

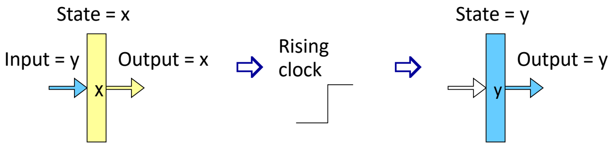
Hardware Registers





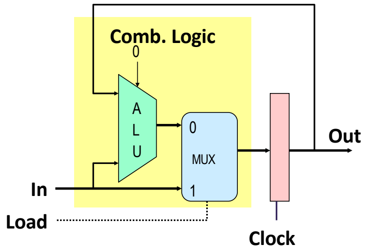
- Stores word of data
- Different from *program registers* seen in assembly code
- Collection of edge-triggered *latches*
- Loads input on *rising edge* of clock

Register Operation

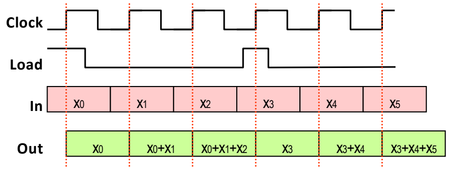


- Register stores data bits
- Acts as barrier between input and output
- As clock rises, loads input, possibly changes output
- Y86-64 processor uses clocked registers to hold PC, CC, and Stat

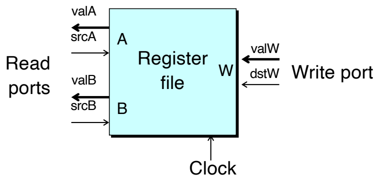
State Machine Example



- Accumulator circuit
- Load or accumulate on each cycle
- (Notice effect of clock on Out)

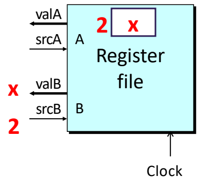


Random-Access Memory

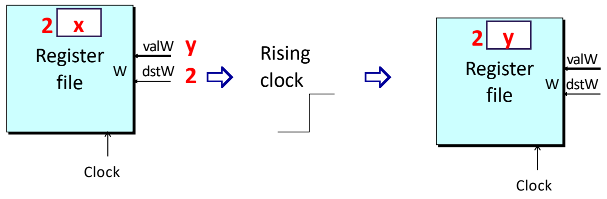


- Stores multiple words of memory (has internal storage)
  - Address input specifies which word to read or write
- Register file
  - Holds values of program registers (`%rax`, `%rsp`, etc.)
  - Register identifier serves as address
    - ID 15 (0xF) implies no read or write performed
- Multiple Ports
  - Can read and/or write multiple words in one cycle
    - Each has separate address and data input/output

Register File Timing



- Reading
  - Like combinational logic
  - Output data generated based on input addr
    - After some small delay
- Writing
  - Like register
  - Update only as clock rises

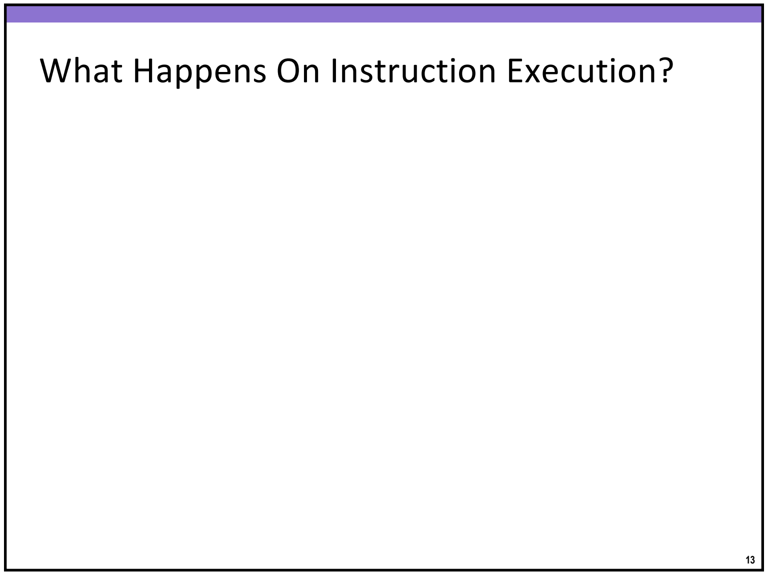


Ch 4.2 Summary

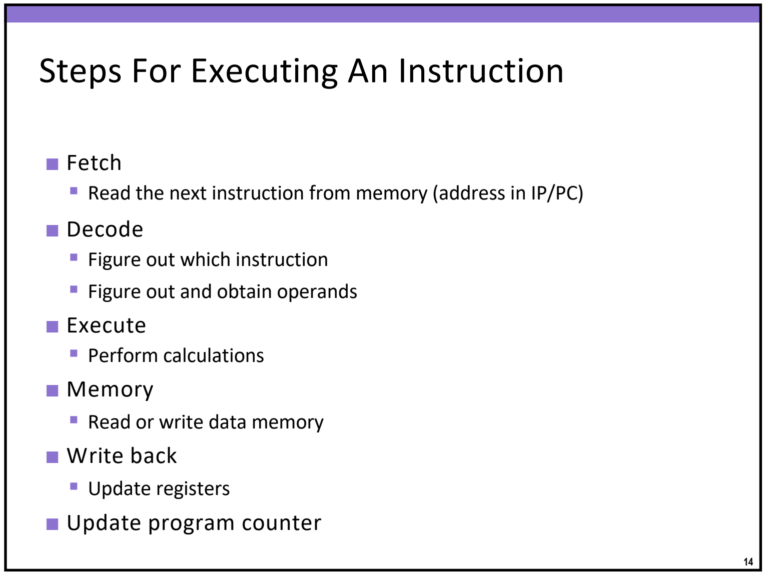
- Computation
  - Performed by combinational logic
  - Computes Boolean functions
  - Continuously reacts to input changes
- Storage
  - Registers (hardware)
    - Hold single words
    - Loaded as clock rises
  - Random-access memories
    - Hold multiple words
    - Possible multiple read or write ports
    - Read word when address input changes
    - Write word as clock rises

Today: The Y86 Datapath

- Memory and clocking
  - How is information stored
- Construction a single-cycle datapath for Y86
- Pipelining Concepts

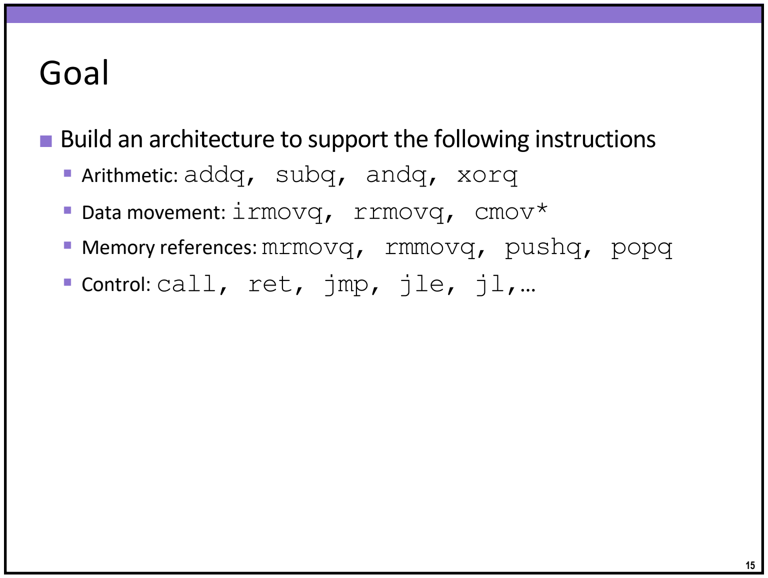


What Happens On Instruction Execution?



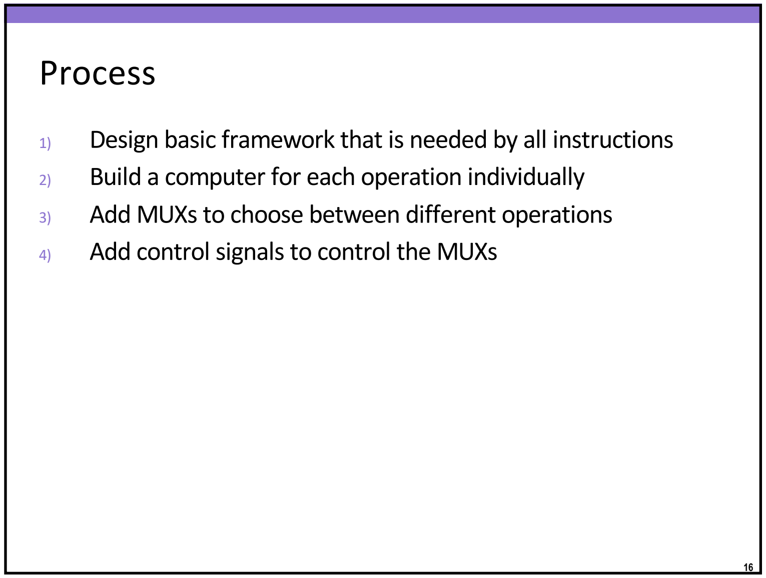
Steps For Executing An Instruction

- Fetch
  - Read the next instruction from memory (address in IP/PC)
- Decode
  - Figure out which instruction
  - Figure out and obtain operands
- Execute
  - Perform calculations
- Memory
  - Read or write data memory
- Write back
  - Update registers
- Update program counter



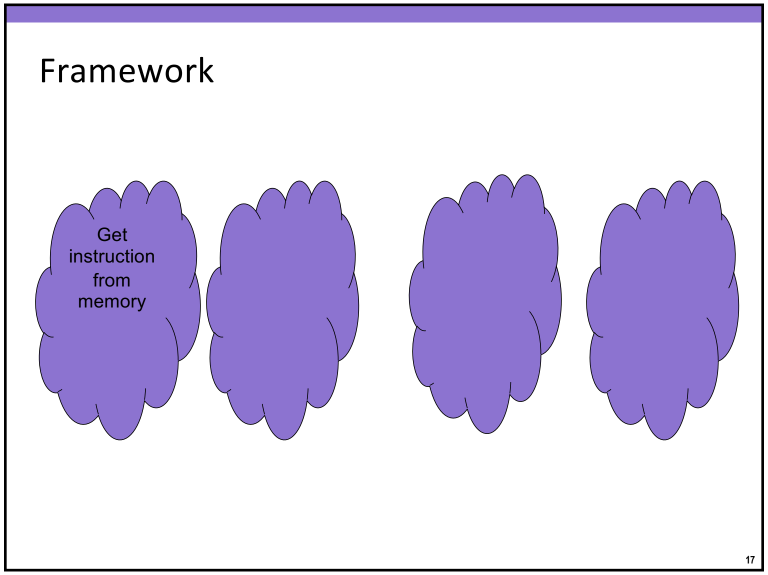
Goal

- Build an architecture to support the following instructions
  - Arithmetic: `addq, subq, andq, xorq`
  - Data movement: `irmovq, rrmovq, cmov*`
  - Memory references: `mrmovq, rmmovq, pushq, popq`
  - Control: `call, ret, jmp, jle, jl,`…



Process

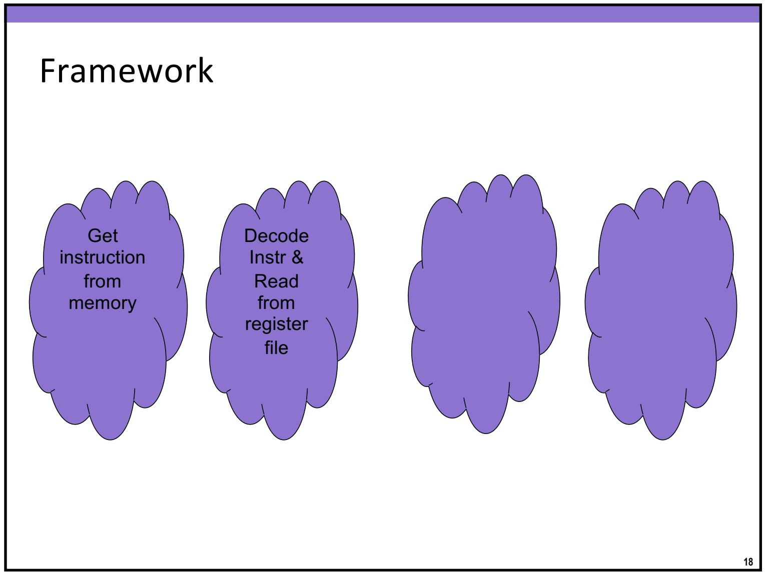
1) Design basic framework that is needed by all instructions
2) Build a computer for each operation individually
3) Add MUXs to choose between different operations
4) Add control signals to control the MUXs



Framework

Get instruction from memory

17

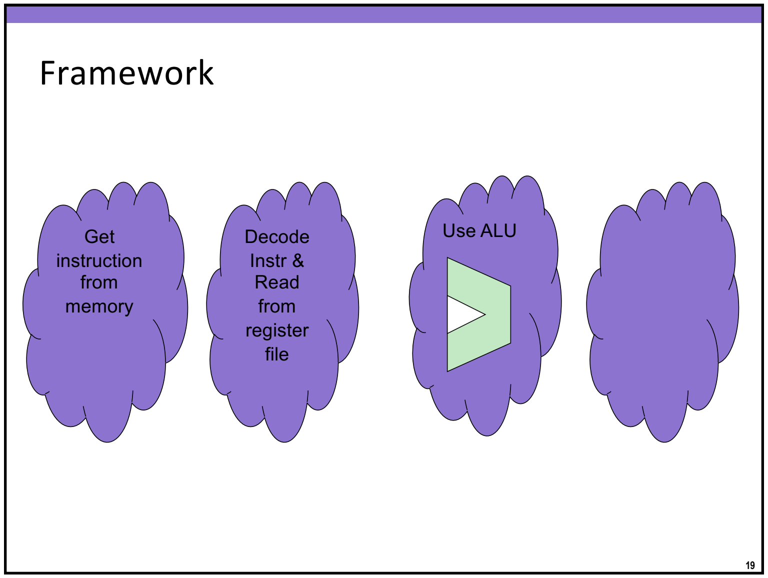


Framework

Get instruction from memory

Decode Instr & Read from register file

18



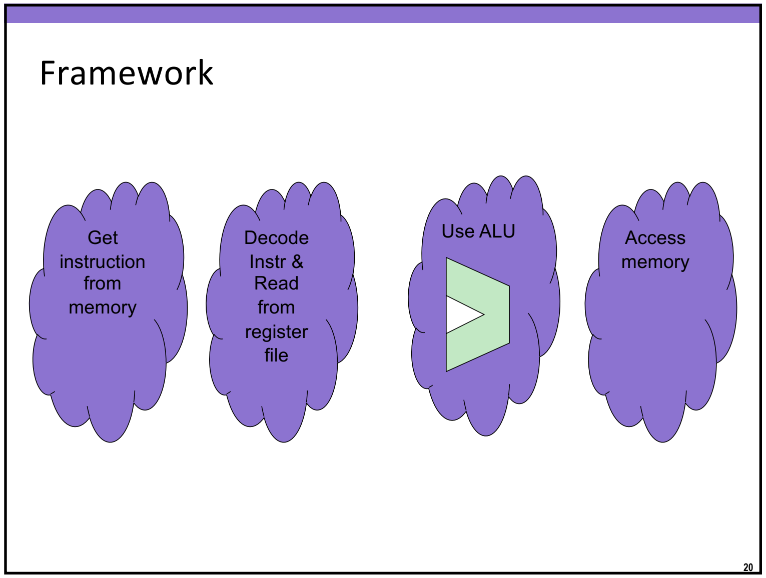
Framework

Get instruction from memory

Decode Instr & Read from register file

Use ALU

19



Framework

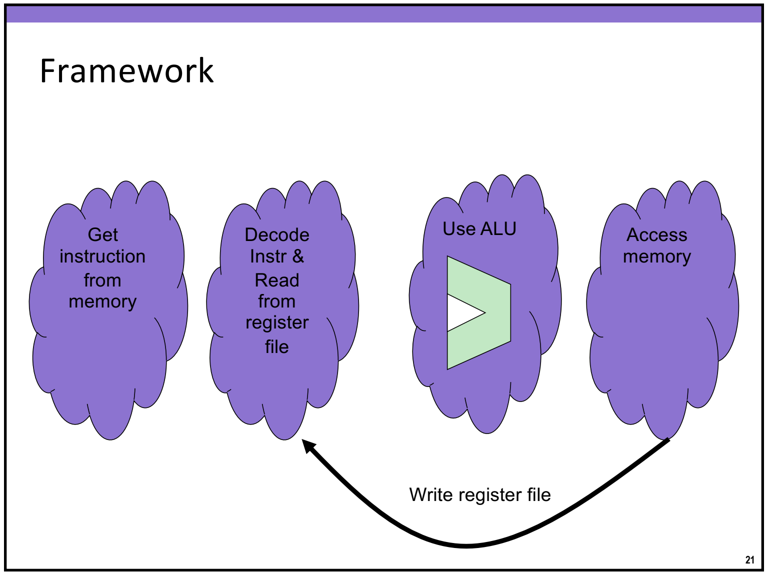
Get instruction from memory

Decode Instr & Read from register file

Use ALU

Access memory

20



Framework

Get instruction from memory

Decode Instr & Read from register file

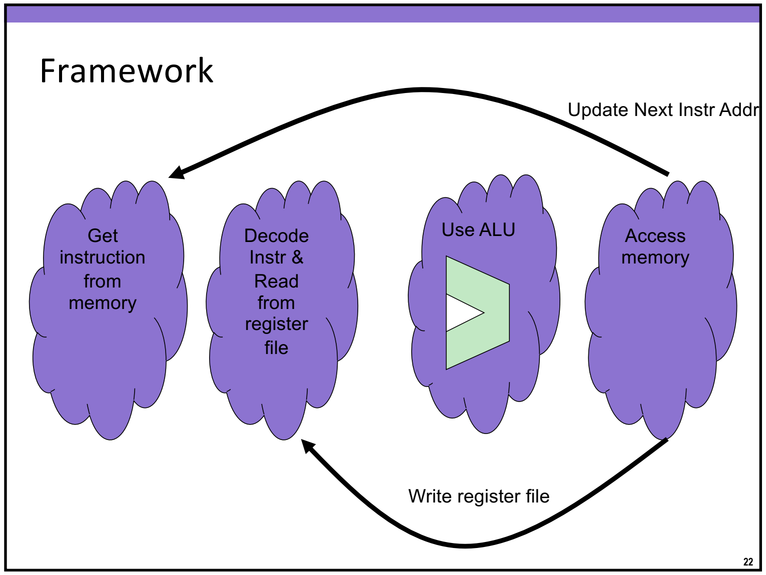
Use ALU

Access memory

Write register file

21

21



Framework

Update Next Instr Addr

Get instruction from memory

Decode Instr & Read from register file

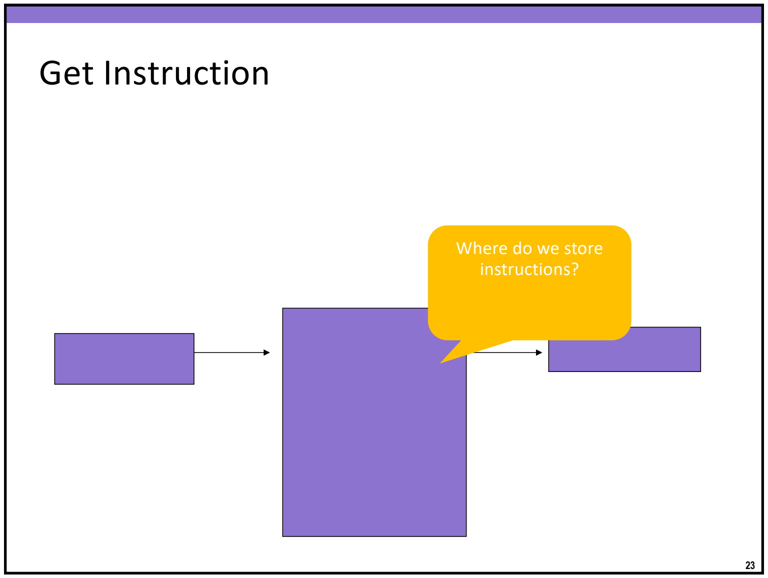
Use ALU

Access memory

Write register file

22

22

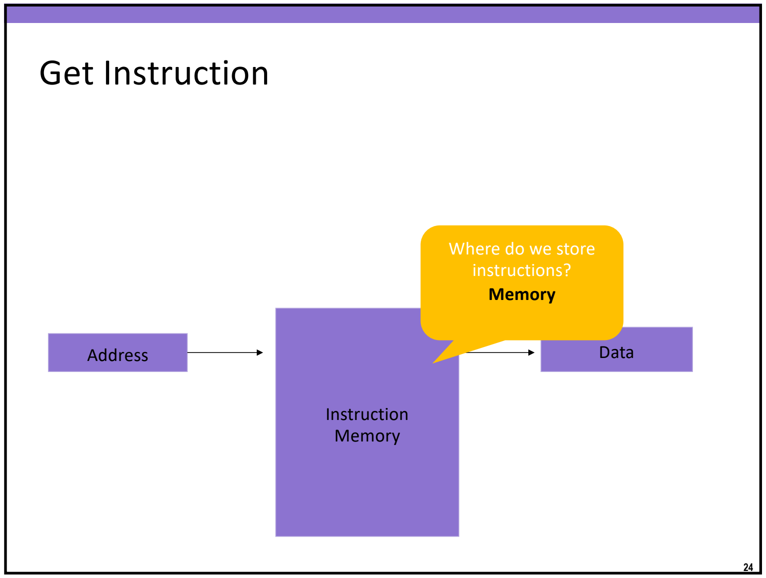


Get Instruction

Where do we store instructions?

23

23



Get Instruction

Where do we store instructions?

Memory

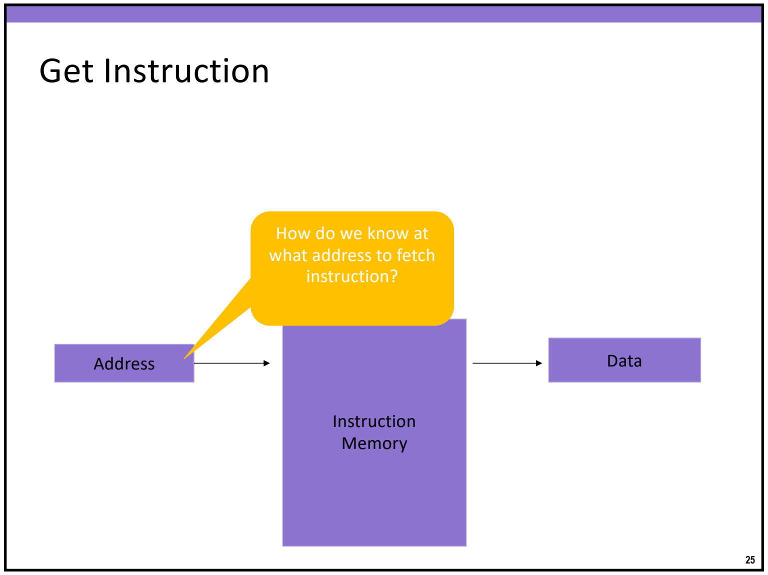
Address

Instruction Memory

Data

24

24



Get Instruction

How do we know at what address to fetch instruction?

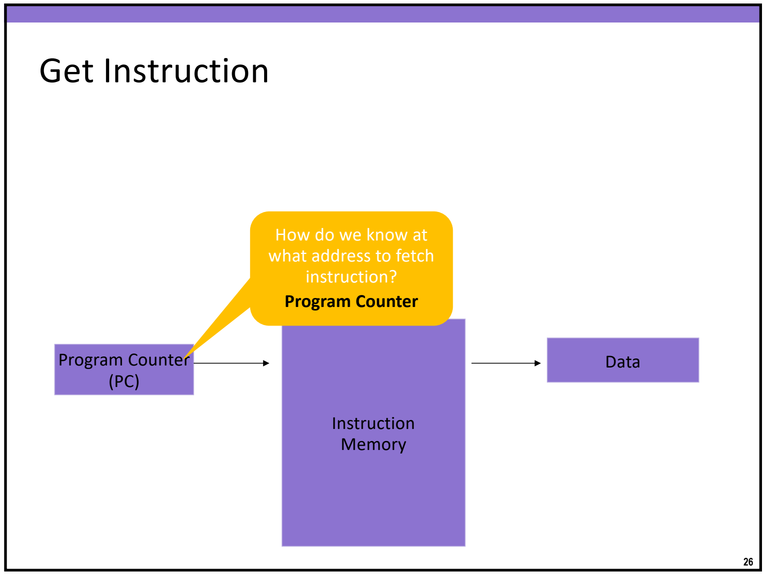
Address

Instruction Memory

Data

25

25



Get Instruction

How do we know at what address to fetch instruction?

Program Counter

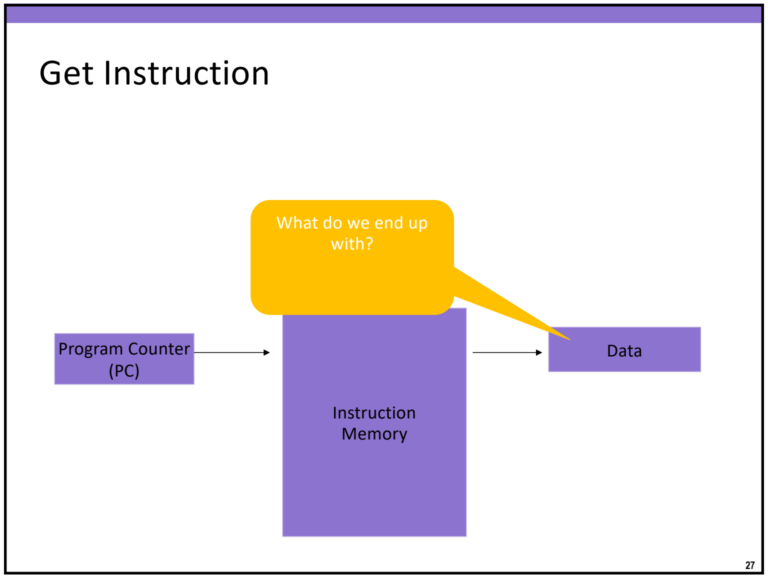
Program Counter (PC)

Instruction Memory

Data

26

26



Get Instruction

What do we end up with?

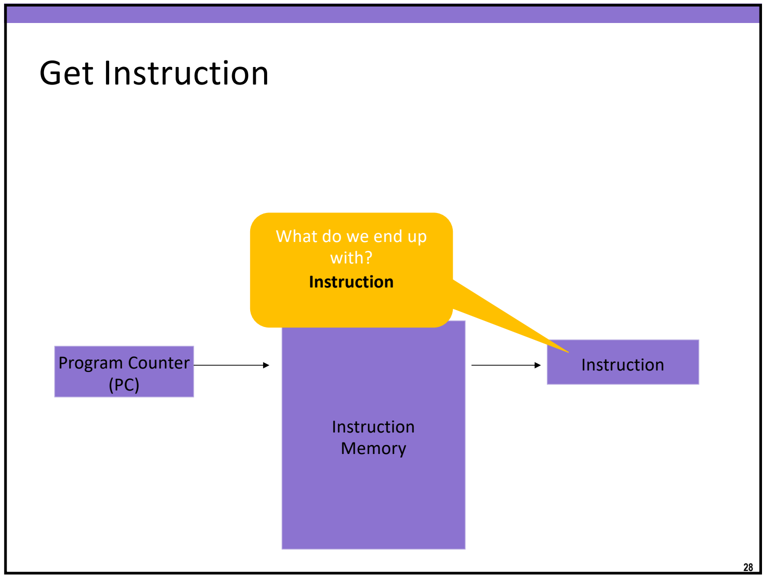
Program Counter (PC)

Instruction Memory

Data

27

27



Get Instruction

What do we end up with?

Instruction

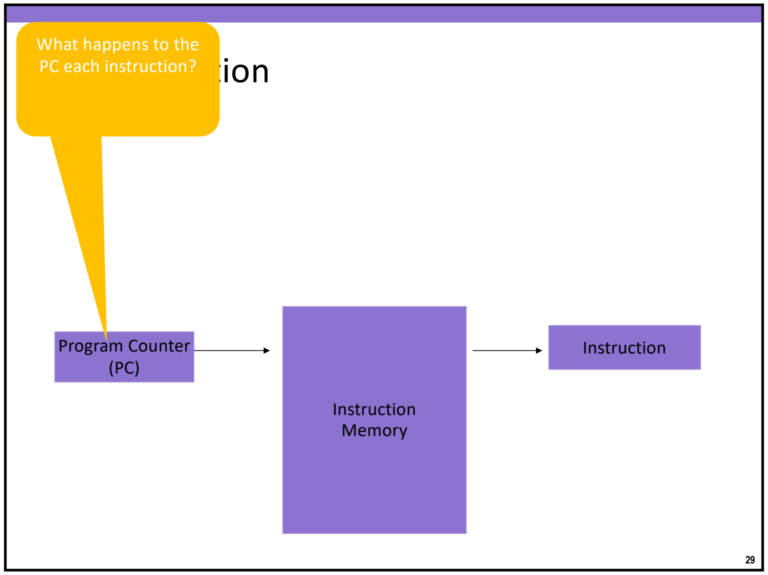
Program Counter (PC)

Instruction Memory

Instruction

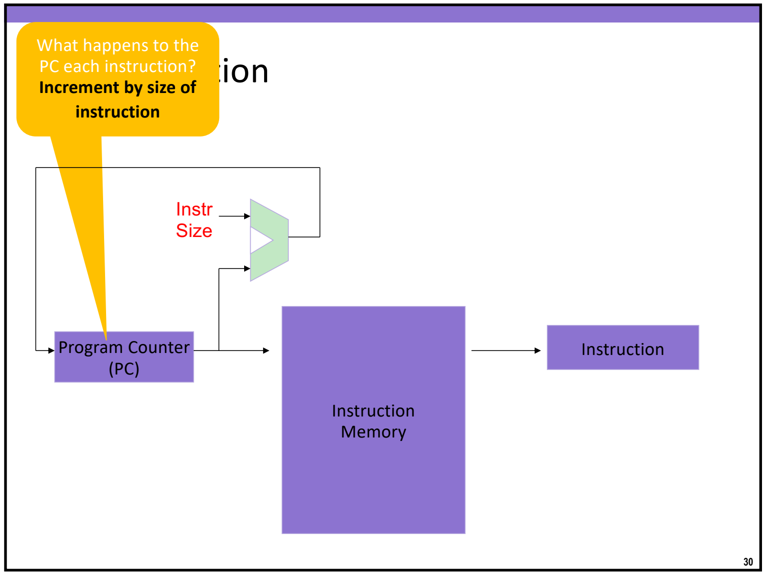
28

28



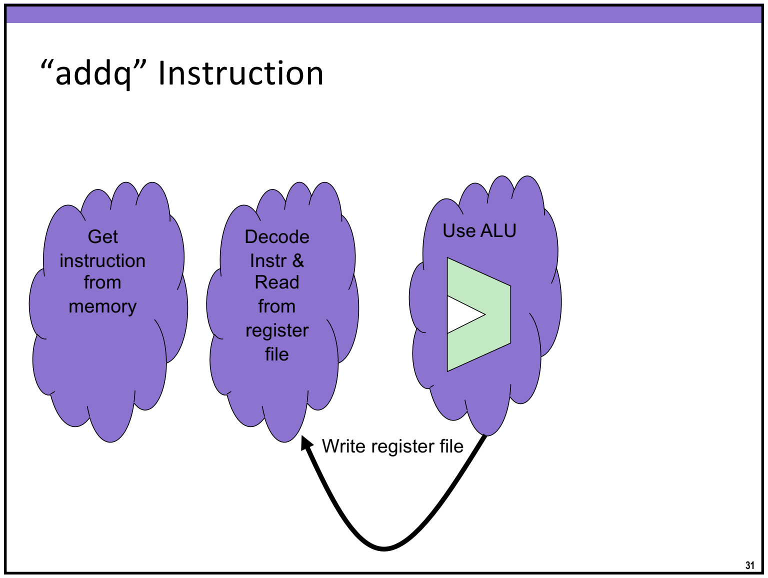
What happens to the PC each instruction?

29



What happens to the PC each instruction? **Increment by size of instruction**

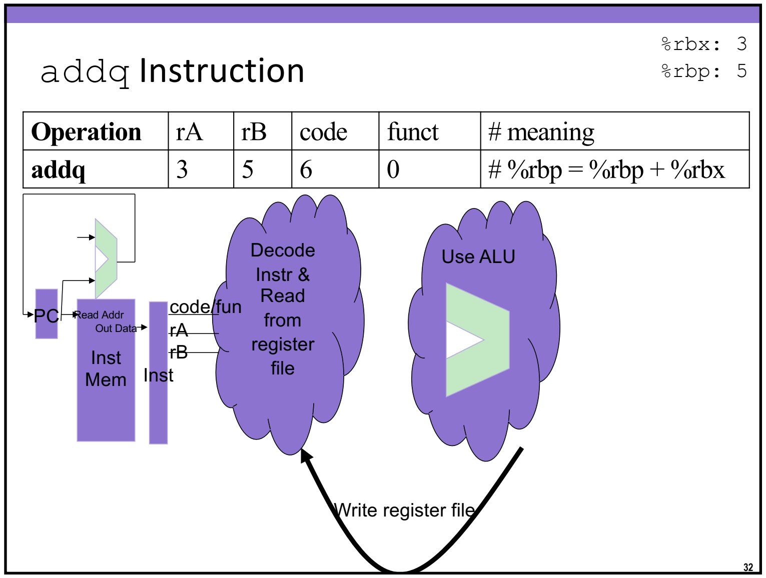
30



“addq” Instruction

Get instruction from memory
Decode Instr & Read from register file
Use ALU
Write register file
31

31

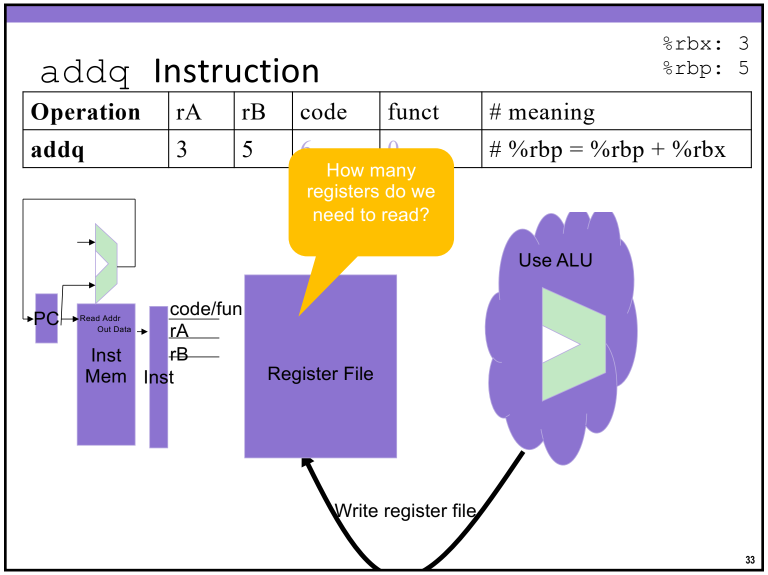


addq Instruction

```
%rbx: 3
%rbp: 5
```

Operation	rA	rB	code	funct	# meaning
addq	3	5	6	0	# %rbp = %rbp + %rbx

32



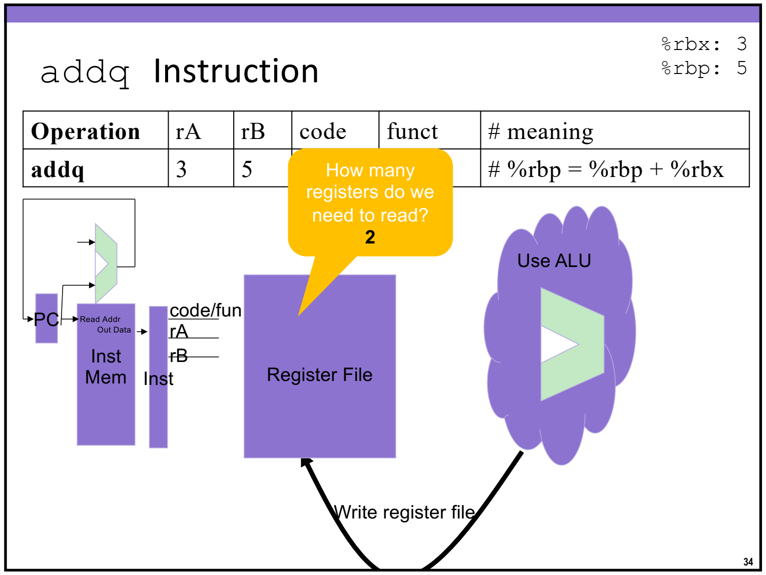
```
%rbx: 3
%rbp: 5
```

addq Instruction

Operation	rA	rB	code	funct	# meaning
addq	3	5	6	0	# %rbp = %rbp + %rbx

How many registers do we need to read?

33



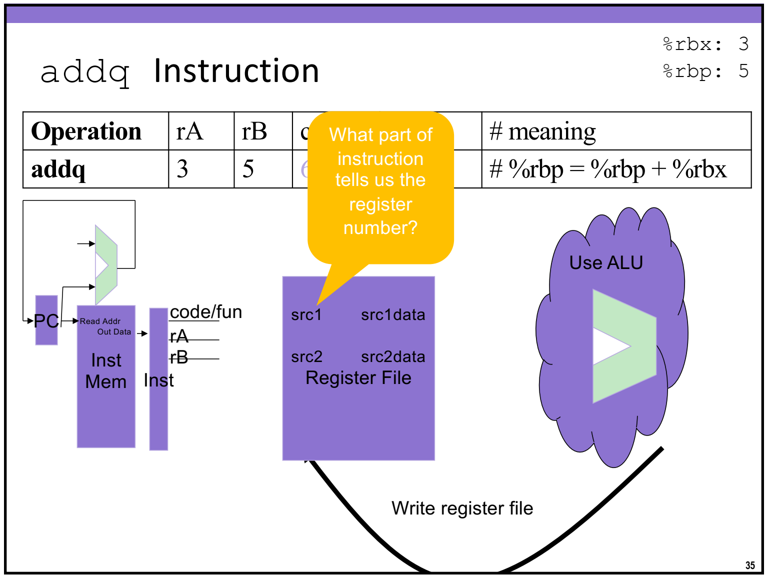
```
%rbx: 3
%rbp: 5
```

addq Instruction

Operation	rA	rB	code	funct	# meaning
addq	3	5			# %rbp = %rbp + %rbx

How many registers do we need to read?
2

34



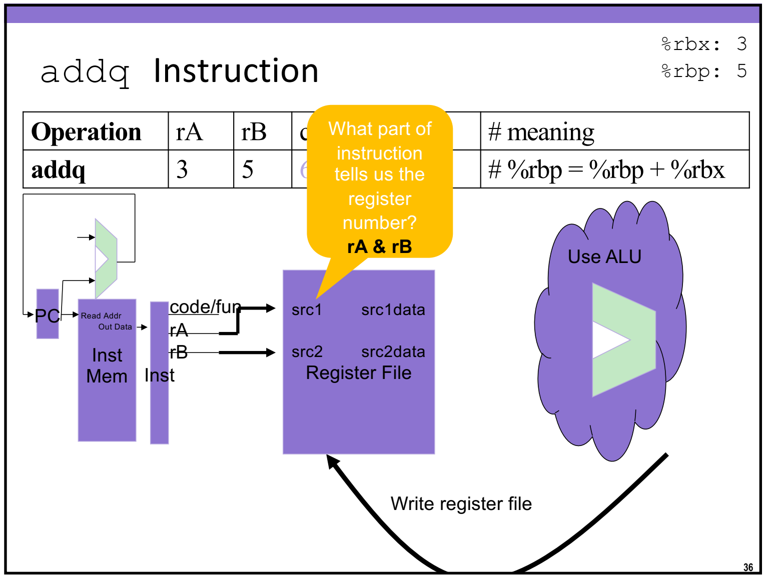
```
%rbx: 3
%rbp: 5
```

addq Instruction

Operation	rA	rB	c	# meaning
addq	3	5	6	# %rbp = %rbp + %rbx

What part of instruction tells us the register number?

35



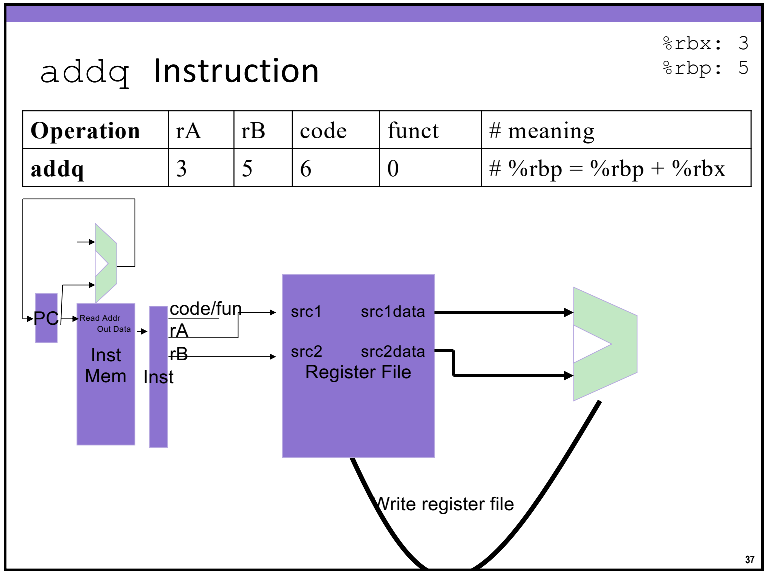
```
%rbx: 3
%rbp: 5
```

addq Instruction

Operation	rA	rB	c	# meaning
addq	3	5	6	# %rbp = %rbp + %rbx

What part of instruction tells us the register number?
rA & rB

36

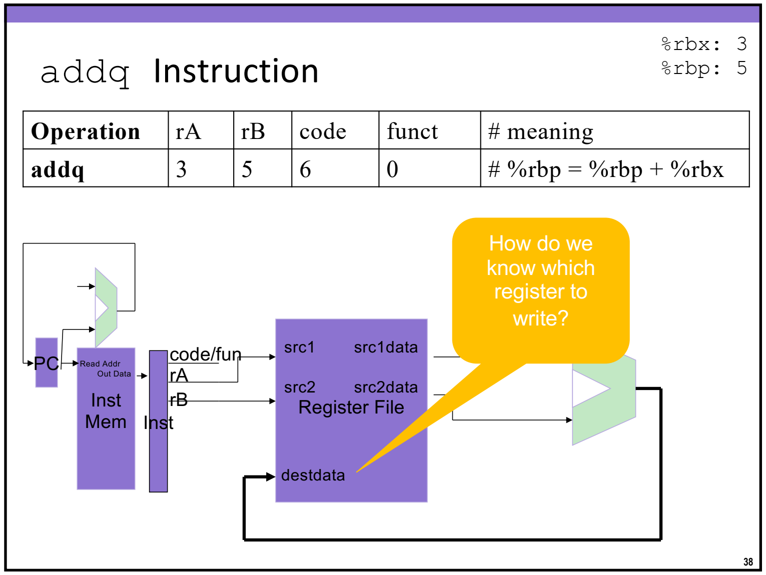


addq Instruction

%rbx: 3
%rbp: 5

Operation	rA	rB	code	funct	# meaning
addq	3	5	6	0	# %rbp = %rbp + %rbx

37

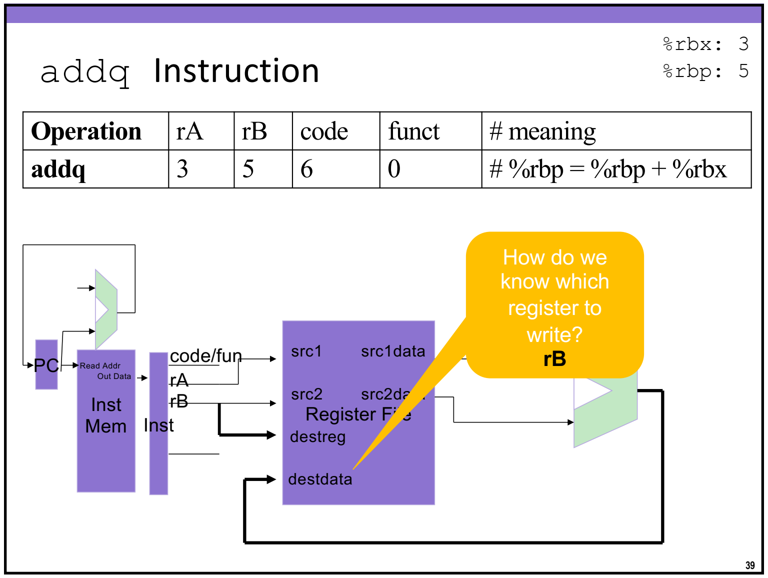


addq Instruction

%rbx: 3
%rbp: 5

Operation	rA	rB	code	funct	# meaning
addq	3	5	6	0	# %rbp = %rbp + %rbx

38

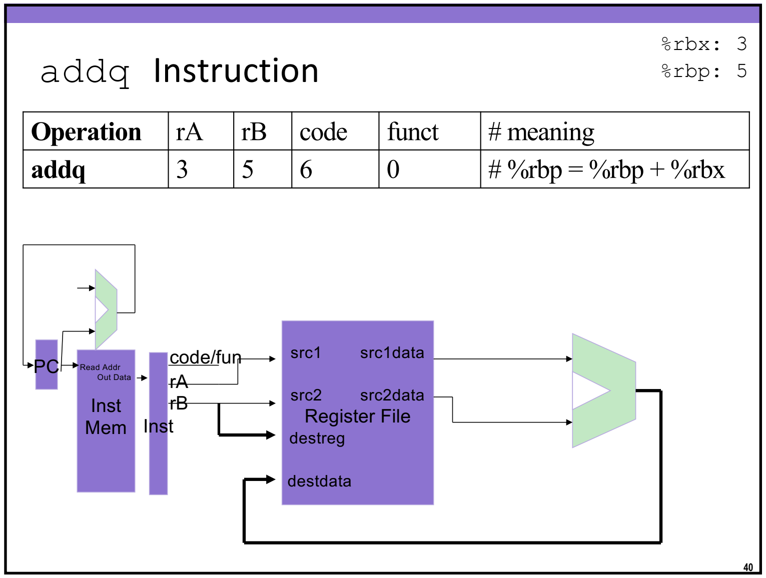


addq Instruction

%rbx: 3
%rbp: 5

Operation	rA	rB	code	funct	# meaning
addq	3	5	6	0	# %rbp = %rbp + %rbx

39

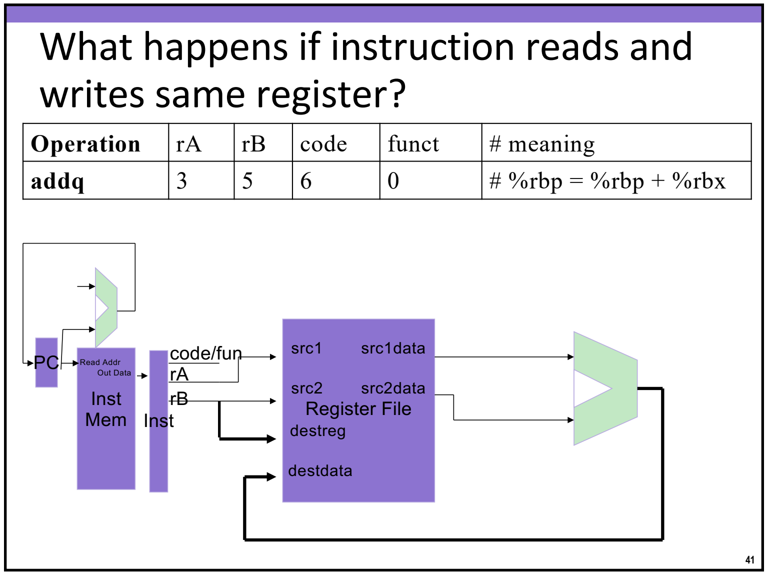


addq Instruction

%rbx: 3
%rbp: 5

Operation	rA	rB	code	funct	# meaning
addq	3	5	6	0	# %rbp = %rbp + %rbx

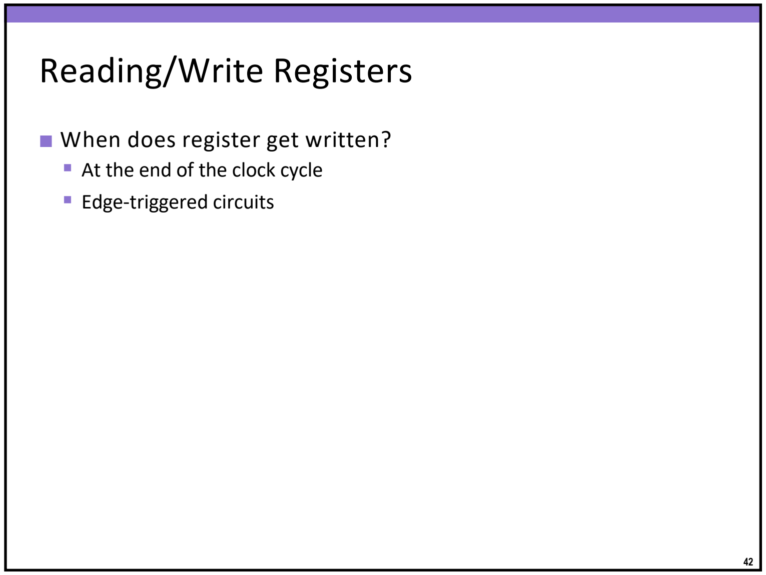
40



What happens if instruction reads and writes same register?

Operation	rA	rB	code	funct	# meaning
addq	3	5	6	0	# %rbp = %rbp + %rbx

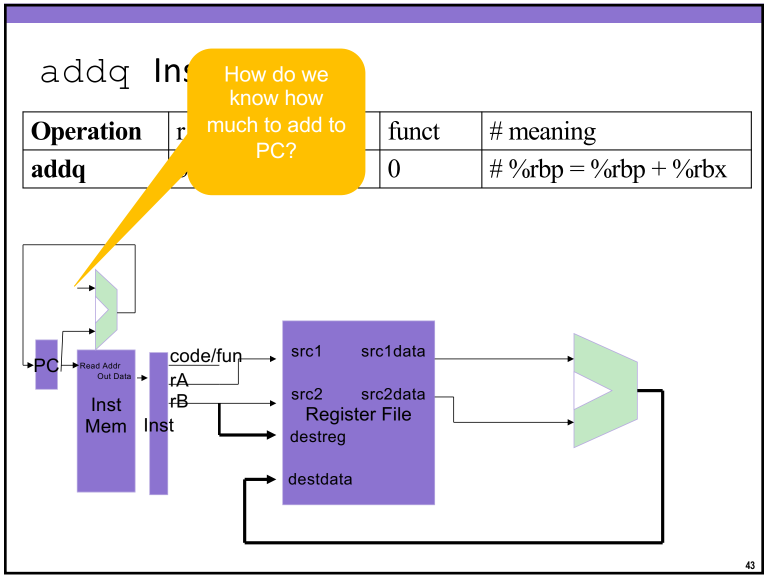
41



Reading/Write Registers

- When does register get written?
  - At the end of the clock cycle
  - Edge-triggered circuits

42

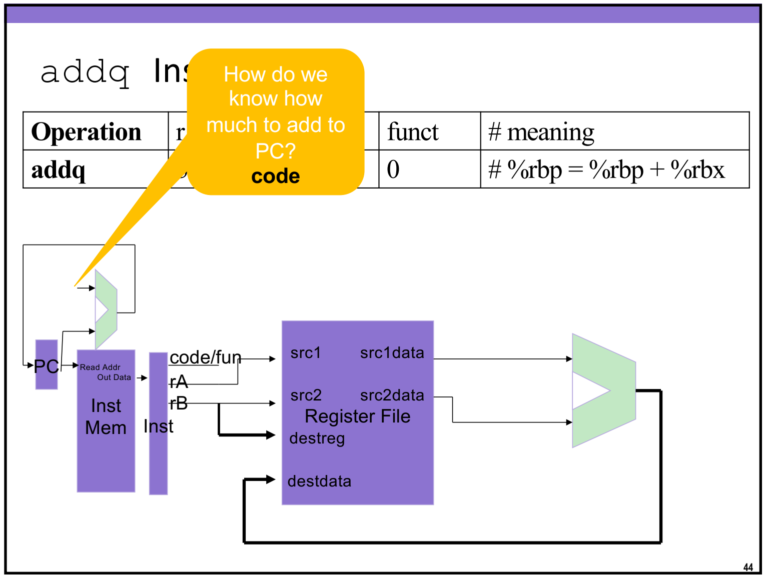


addq Ins

How do we know how much to add to PC?

Operation	r	funct	# meaning
addq		0	# %rbp = %rbp + %rbx

43

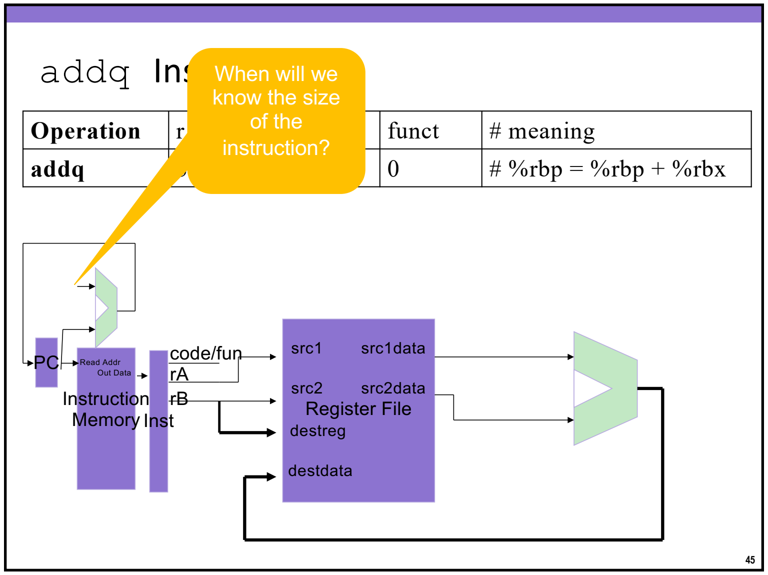


addq Ins

How do we know how much to add to PC?
code

Operation	r	funct	# meaning
addq		0	# %rbp = %rbp + %rbx

44

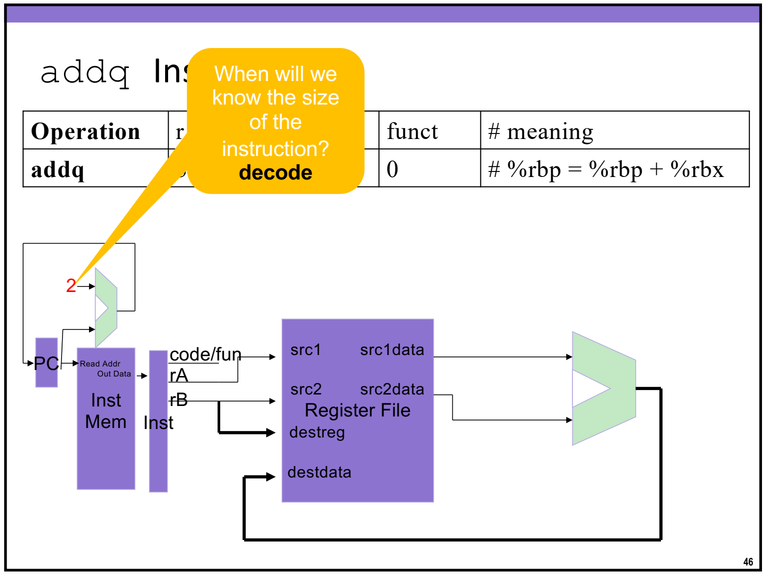


addq Ins

Operation	r	funct	# meaning
addq		0	# %rbp = %rbp + %rbx

When will we know the size of the instruction?

45

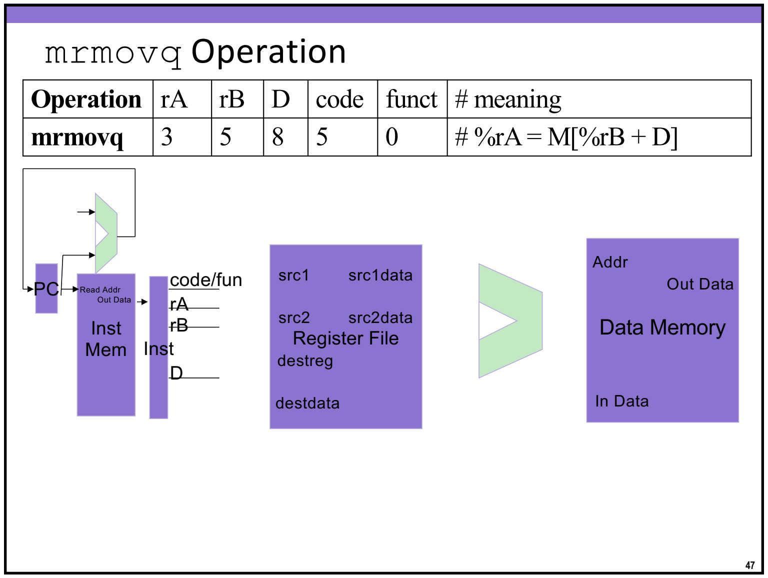


addq Ins

Operation	r	funct	# meaning
addq		0	# %rbp = %rbp + %rbx

When will we know the size of the instruction? **decode**

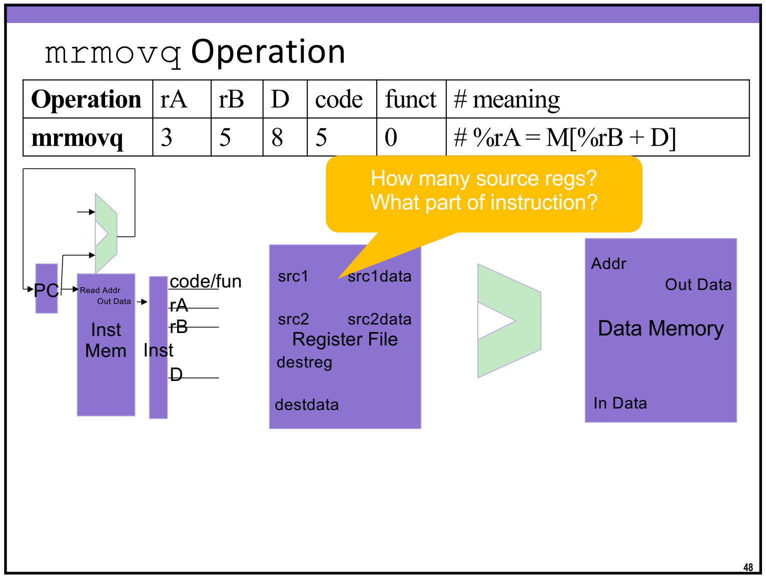
46



mrmovq Operation

Operation	rA	rB	D	code	funct	# meaning
mrmovq	3	5	8	5	0	# %rA = M[%rB + D]

47

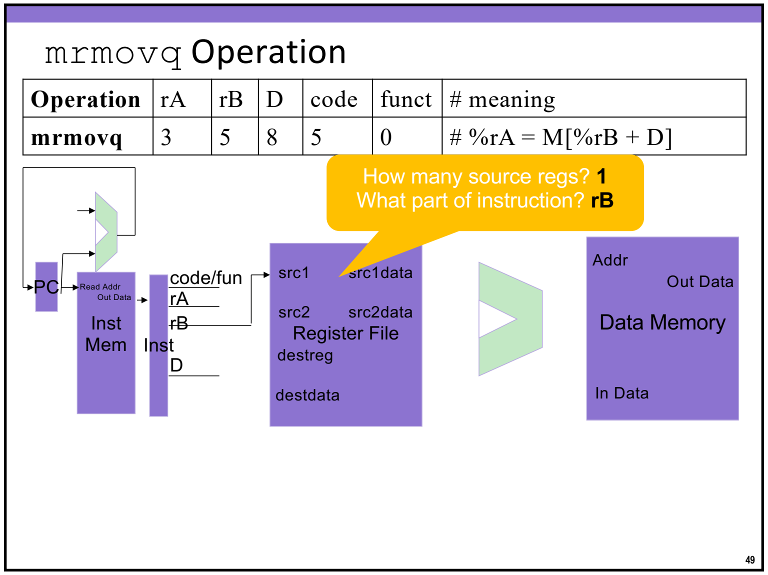


mrmovq Operation

Operation	rA	rB	D	code	funct	# meaning
mrmovq	3	5	8	5	0	# %rA = M[%rB + D]

How many source regs? What part of instruction?

48

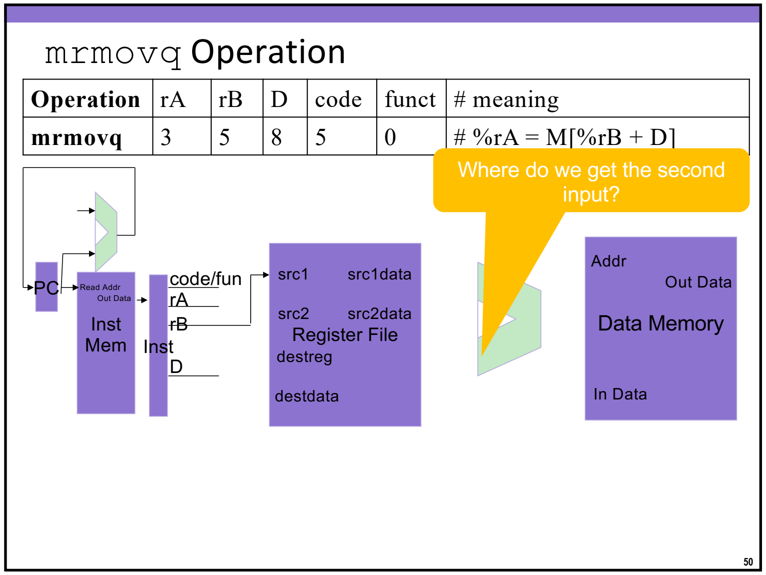


mrmovq Operation

Operation	rA	rB	D	code	funct	# meaning
mrmovq	3	5	8	5	0	# %rA = M[%rB + D]

How many source regs? **1**
What part of instruction? **rB**

49

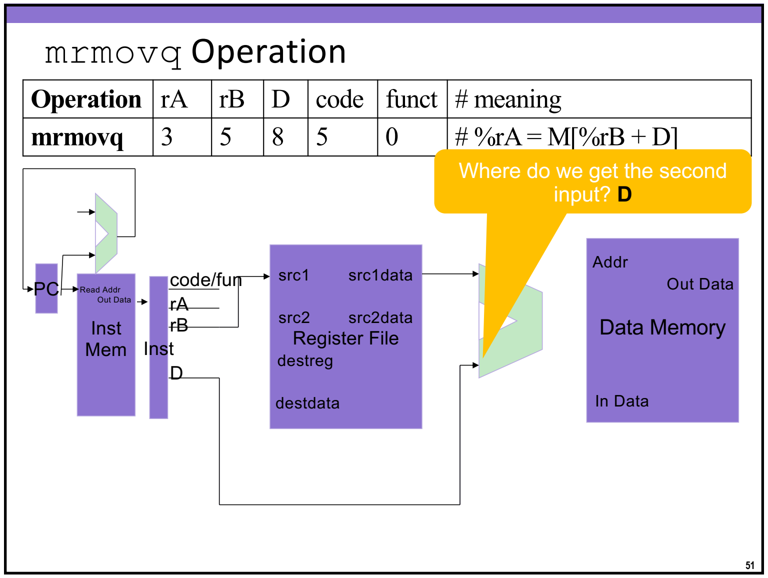


mrmovq Operation

Operation	rA	rB	D	code	funct	# meaning
mrmovq	3	5	8	5	0	# %rA = M[%rB + D]

Where do we get the second input?

50

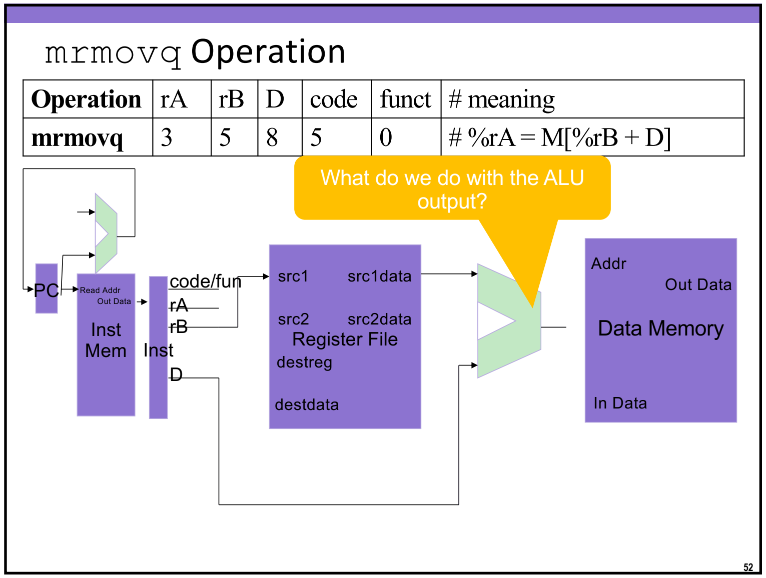


mrmovq Operation

Operation	rA	rB	D	code	funct	# meaning
mrmovq	3	5	8	5	0	# %rA = M[%rB + D]

Where do we get the second input? **D**

51



mrmovq Operation

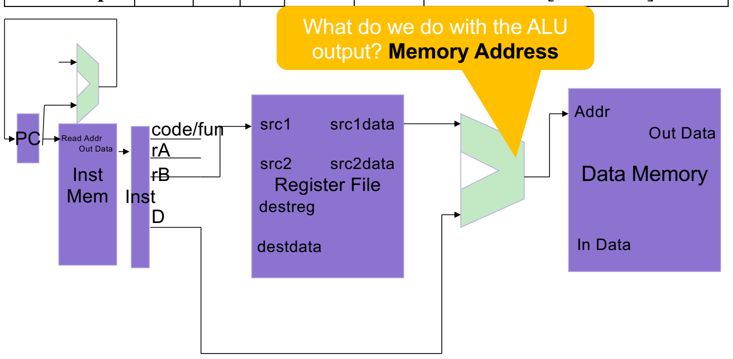
Operation	rA	rB	D	code	funct	# meaning
mrmovq	3	5	8	5	0	# %rA = M[%rB + D]

What do we do with the ALU output?

52

mrmovq Operation

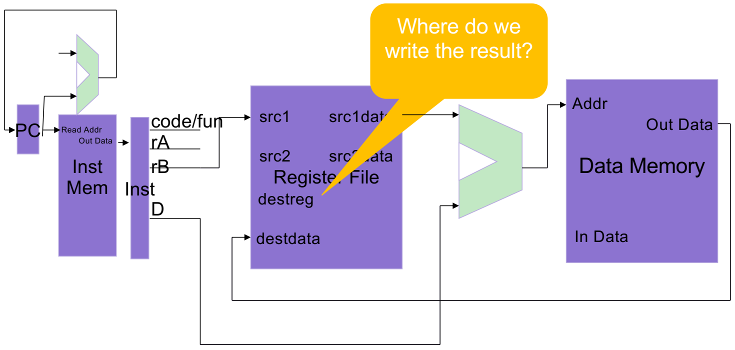
Operation	rA	rB	D	code	funct	# meaning
mrmovq	3	5	8	5	0	# %rA = M[%rB + D]



What do we do with the ALU output? **Memory Address**

mrmovq Operation

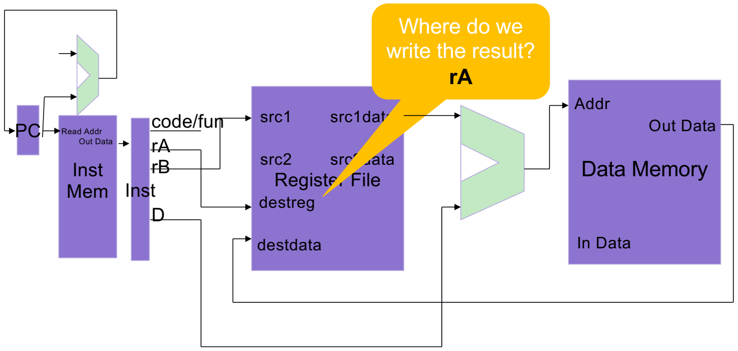
Operation	rA	rB	D	code	funct	# meaning
mrmovq	3	5	8	5	0	# %rA = M[%rB + D]



Where do we write the result?

mrmovq Operation

Operation	rA	rB	D	code	funct	# meaning
mrmovq	3	5	8	5	0	# %rA = M[%rB + D]

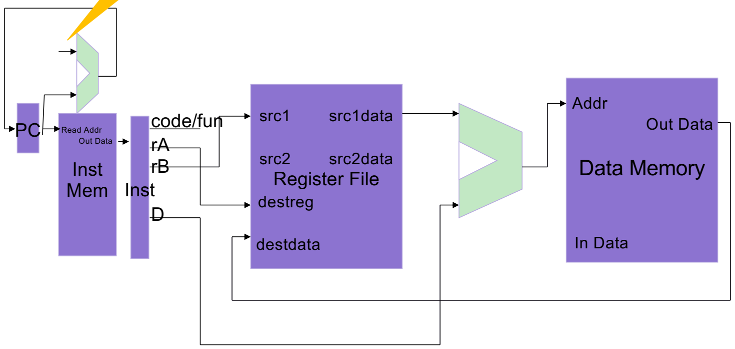


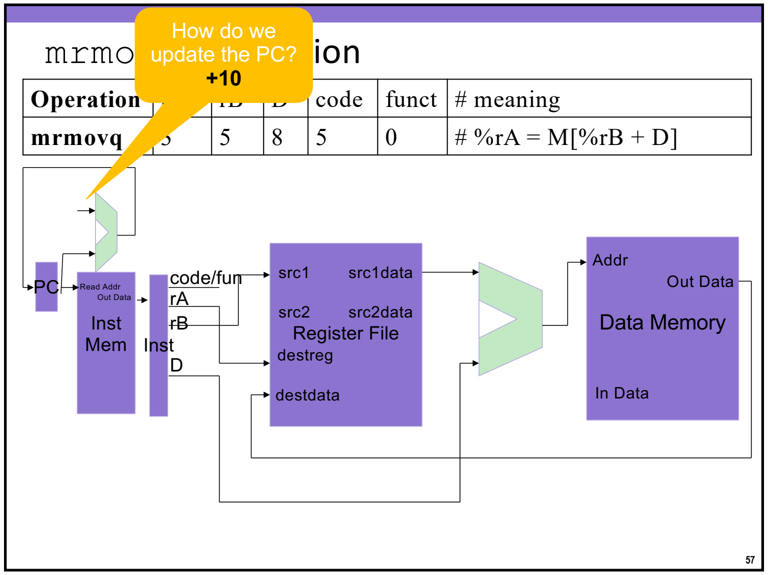
Where do we write the result? **rA**

mrmo ation

How do we update the PC?

Operation	rA	rB	D	code	funct	# meaning
mrmovq	3	5	8	5	0	# %rA = M[%rB + D]



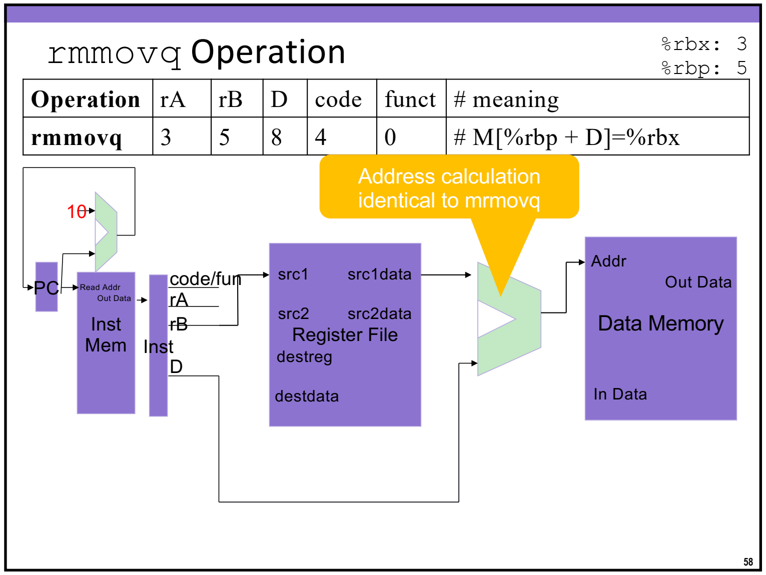


mrmo ion

How do we update the PC? **+10**

Operation	rA	rB	D	code	funct	# meaning
mrmovq	3	5	8	5	0	# %rA = M[%rB + D]

57



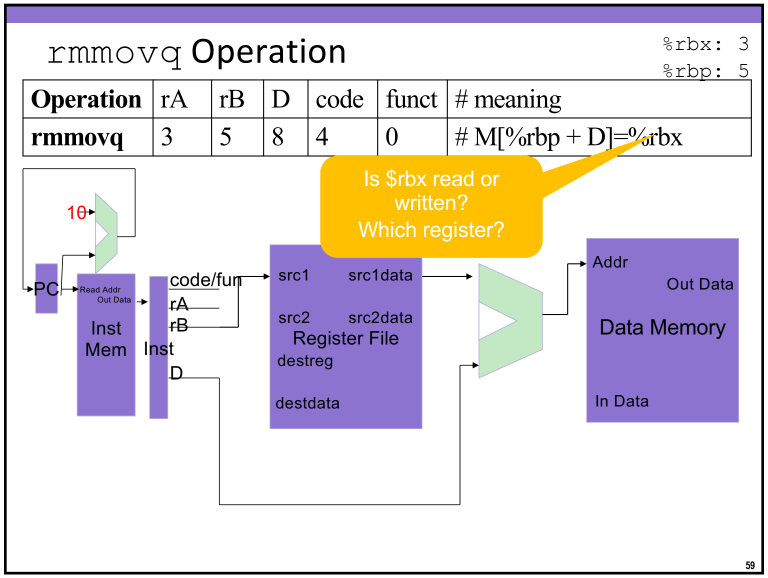
rmmovq Operation

%rbx: 3
%rbp: 5

Operation	rA	rB	D	code	funct	# meaning
rmmovq	3	5	8	4	0	# M[%rbp + D]=%rbx

Address calculation identical to mrmovq

58



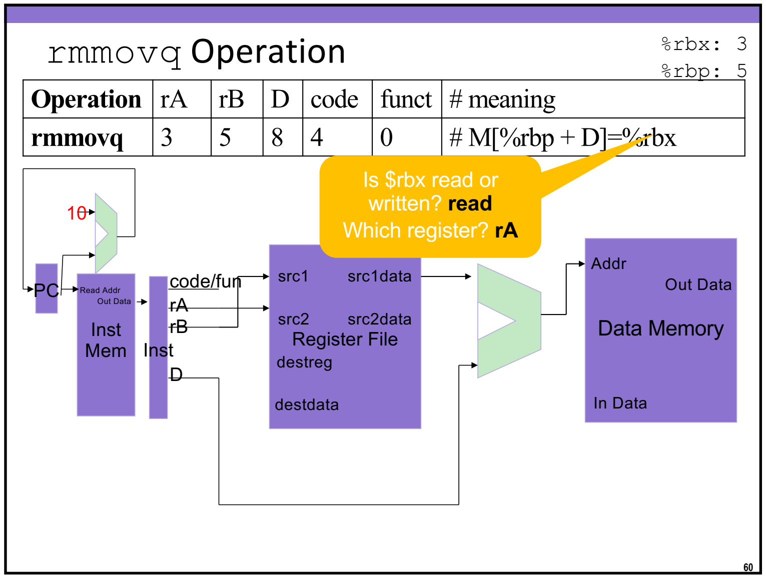
rmmovq Operation

%rbx: 3
%rbp: 5

Operation	rA	rB	D	code	funct	# meaning
rmmovq	3	5	8	4	0	# M[%rbp + D]=%rbx

Is $rbx read or written?
Which register?

59



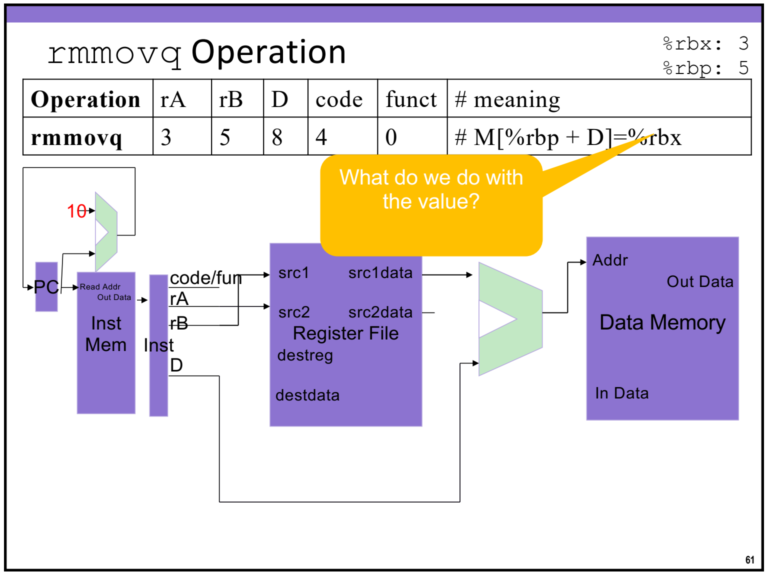
rmmovq Operation

%rbx: 3
%rbp: 5

Operation	rA	rB	D	code	funct	# meaning
rmmovq	3	5	8	4	0	# M[%rbp + D]=%rbx

Is $rbx read or written? **read**
Which register? **rA**

60



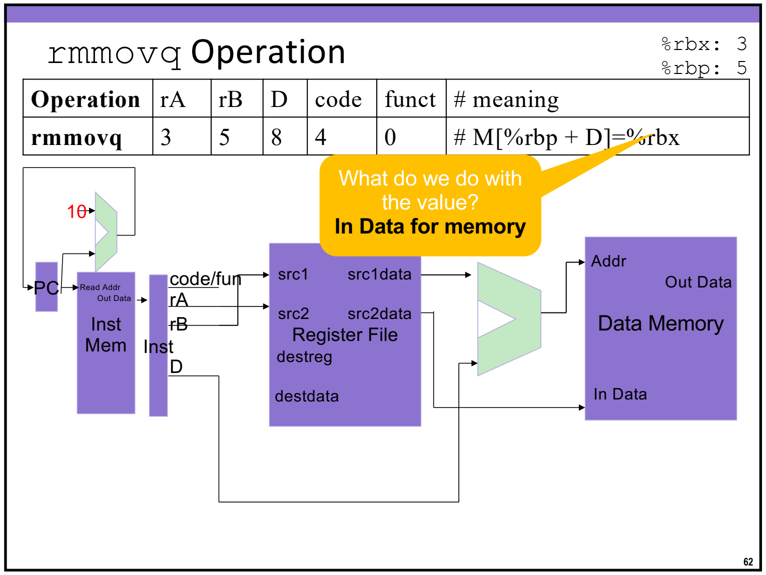
rmmovq Operation

%rbx: 3
%rbp: 5

Operation	rA	rB	D	code	funct	# meaning
rmmovq	3	5	8	4	0	# M[%rbp + D]=%rbx

What do we do with the value?

61



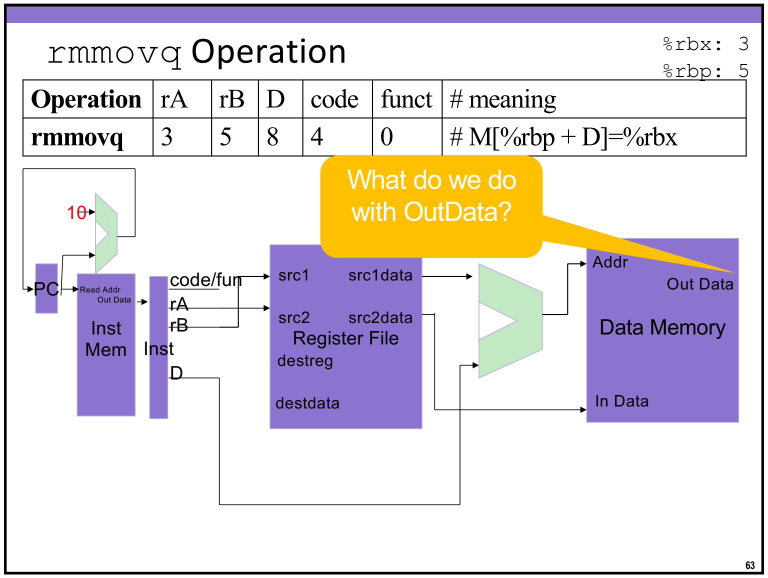
rmmovq Operation

%rbx: 3
%rbp: 5

Operation	rA	rB	D	code	funct	# meaning
rmmovq	3	5	8	4	0	# M[%rbp + D]=%rbx

What do we do with the value?
In Data for memory

62



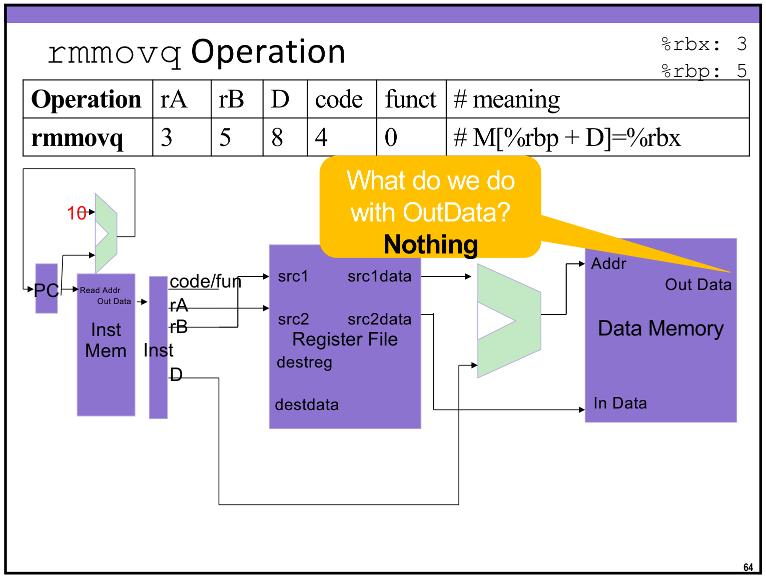
rmmovq Operation

%rbx: 3
%rbp: 5

Operation	rA	rB	D	code	funct	# meaning
rmmovq	3	5	8	4	0	# M[%rbp + D]=%rbx

What do we do with OutData?

63



rmmovq Operation

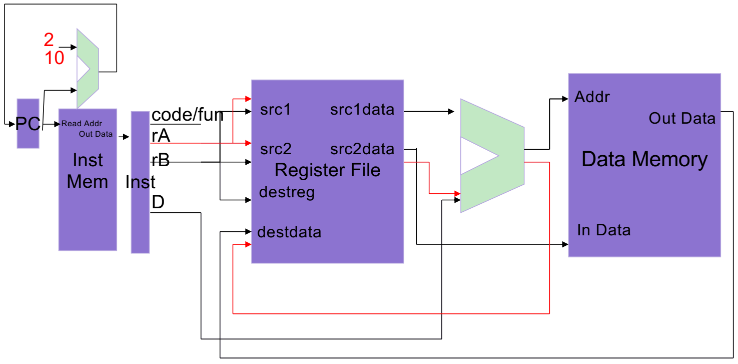
%rbx: 3
%rbp: 5

Operation	rA	rB	D	code	funct	# meaning
rmmovq	3	5	8	4	0	# M[%rbp + D]=%rbx

What do we do with OutData?
Nothing

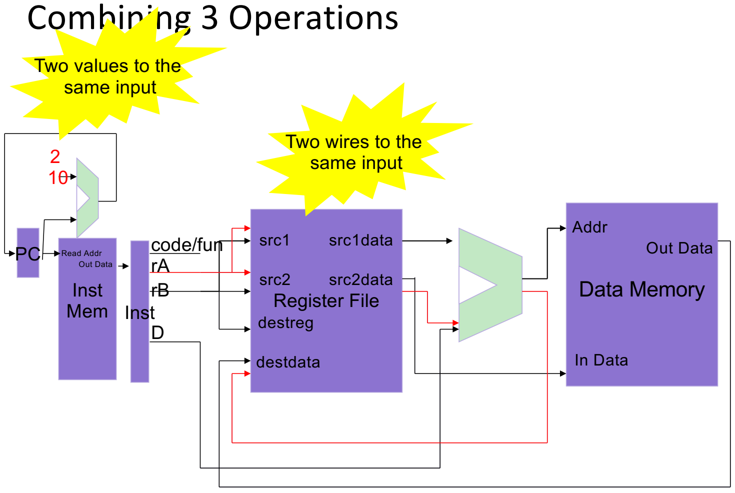
64

Combining 3 Operations



65

Combining 3 Operations

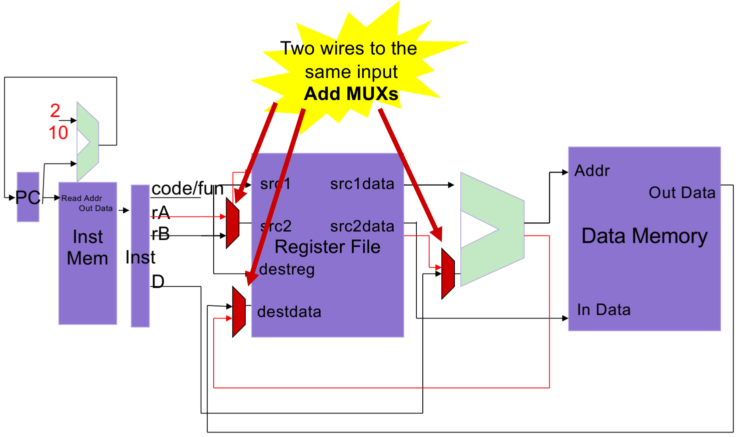


Two values to the same input

Two wires to the same input

66

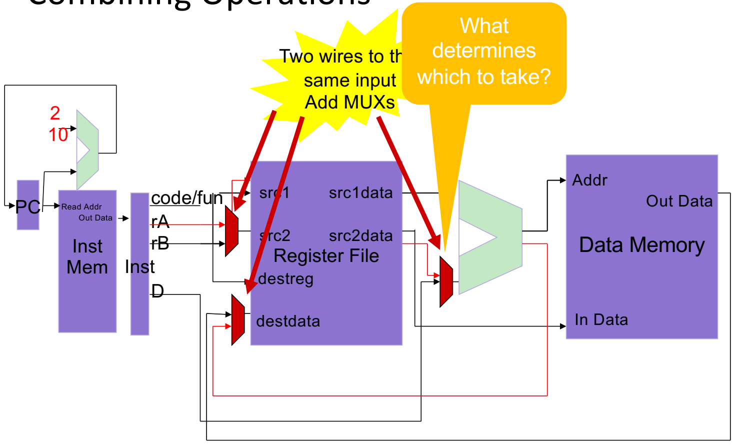
Combining 3 Operations



Two wires to the same input **Add MUXs**

67

Combining Operations



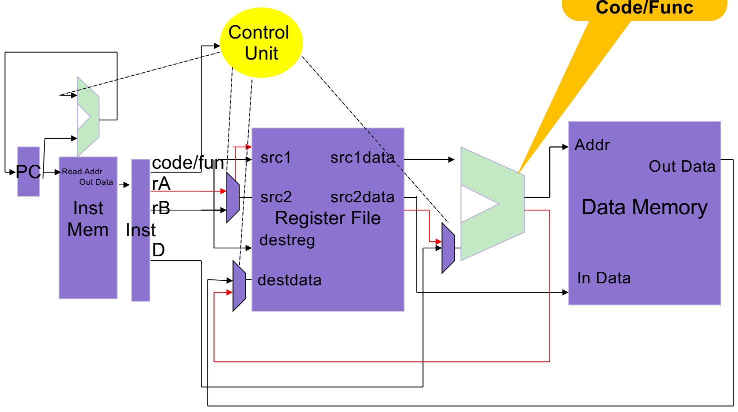
Two wires to th same input Add MUXs

What determines which to take?

68

Combining Operations

What determines which to take?
Code/Func

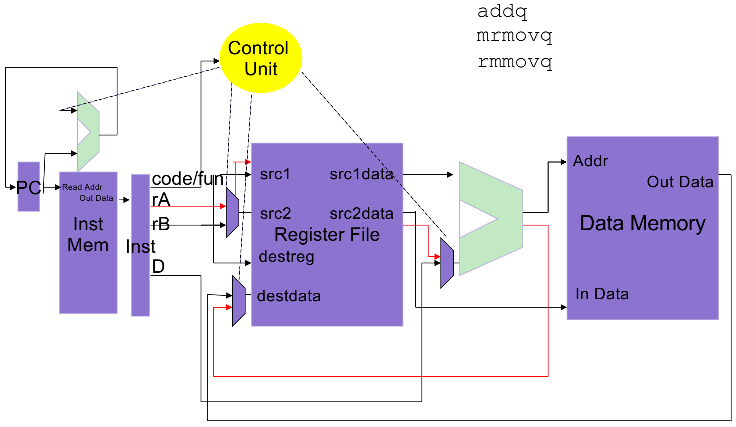


69

69

Combining Operations

```
addq
mrmovq
rmmovq
```



70

70