Instruction Set Architecture

Choosing Ingredients for a Computer Design

סקירת הפרק

מהו מעבד

- מבנה von Neumann
- שלבי תכנון מעבד

קבוצת הפקודות

- מבנה פקודה
- אופרנדים (נתונים)
- שמירת נתונים וסוגי זכרון
- פעולות

שיקולים לתכנון קבוצת הפקודות

- שפה מורכבת (CISC)

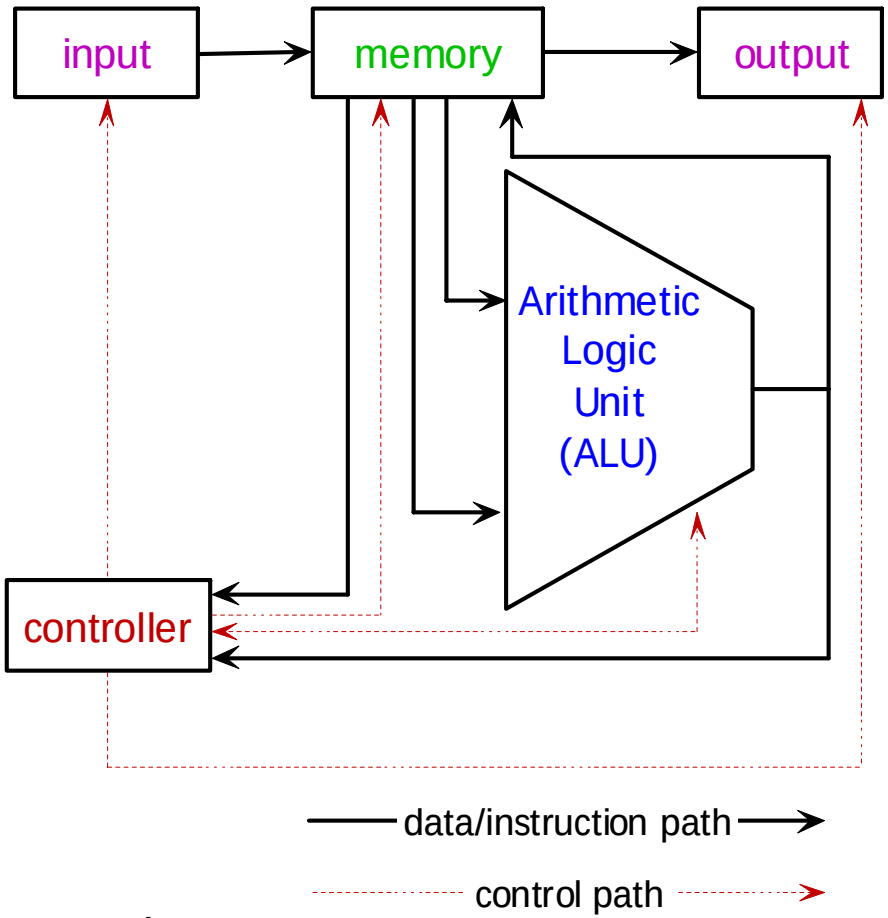
מימוש פקודות בחומרה

- microcode

Von Neumann Architecture

Stored-Program Digital Computer

- Digital computation in **ALU**
- **Programmable** via set of standard instructions
- Internal **storage** of data
- Internal **storage** of program
- Automatic **Input/Output**
- Automatic **sequencing** of instruction execution by decoder/controller



Von Neumann Architecture

Data and instructions stored in a single memory unit

Harvard Architecture

Data and instructions stored in a separate memory units

Stages in Computer Design

Instruction Set Architecture (ISA)

1. Look at universe of problems to be solved
2. Define atomic operations at level of system programmer
   - Small and orthogonal operations (each performs different task)
   - Can be combined to perform any operation
3. Specify instruction set for machine language
   - Choose a minimum set of basic operations
   - Not too many ways to solve same problem

Implementation

1. Design machine as implementation of ISA
2. Evaluate theoretical performance
3. Identify performance problem areas
4. Improve processor efficiency

Instruction Features

Instruction

Description of an **Operation** performed on **Operands**

Operations

Actions performed on data

Operands

Sources — data inputs to operation

Destinations — data outputs from operation

Specified by

Addressing Mode — location of data in machine

Data Type —Integer, Long, Floating Point, Decimal, String, Constant, etc.

Instruction Set Architecture

General Instruction is instance of data structure



Machine Language is range of data structure Instruction for

Operation ∈ {legal actions}

Operand ∈ {legal Addressing Modes}

Describe sources and destinations

Typical machine instruction

```
ADD destination, source_1, source_2
destination ← source_1 + source_2
```

Instruction Definitions

Operations and operands

unary — one source operand

binary — two source operands

n-ary — n source operands

Address specifier

Describes address format

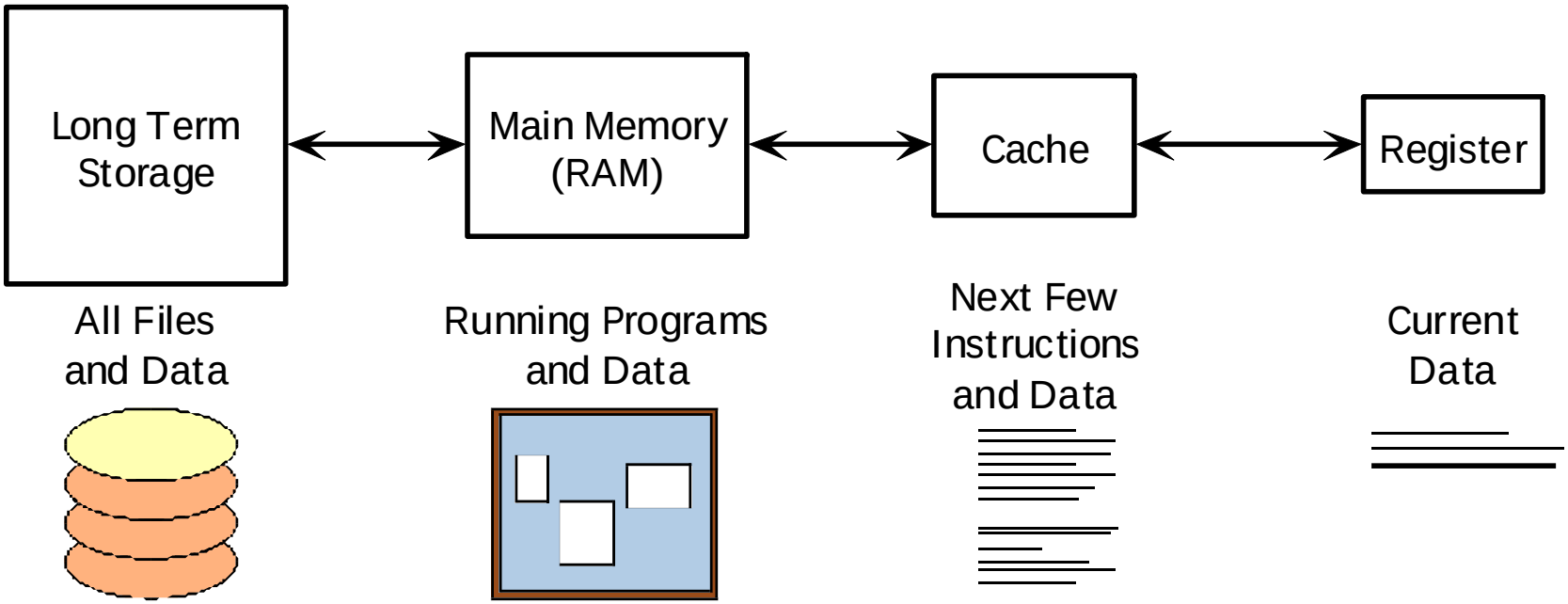
- Addressing mode
- Operation model

Data width

	Intel	Non-Intel
2 bytes	word	half-word
4 bytes	dword	word
8 bytes	quadword	doubleword

Memory Hierarchy

Memory locations outside CPU and RAM	Memory location outside CPU	Memory location in or near CPU	Memory location inside CPU
Stores data and instructions of "all" programs	Stores "all" data and instructions of running programs	Fast access to important data and instructions from RAM	Fast access to small amount of information
Organized by OS	Organized by addresses	Copy of RAM section	Organized by CPU



Register Naming

Registers are part of CPU design

- Information stored in registers called architectural state
- Describes machine status and program status

General Purpose (GP) registers

- Hold data for instructions
  - Width of data is width of standard integer in CPU
- Referenced by names or numbers
  - Intel x86: **`EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP, EIP`**
  - General: **`R0, R1, … , R127`**

Special Purpose registers

- Machine status registers
- Operating system registers

Flat Memory Organization

N-bit address space

Physical Address = $A_{N-1}\, A_{N-2} \ldots A_1\, A_0$

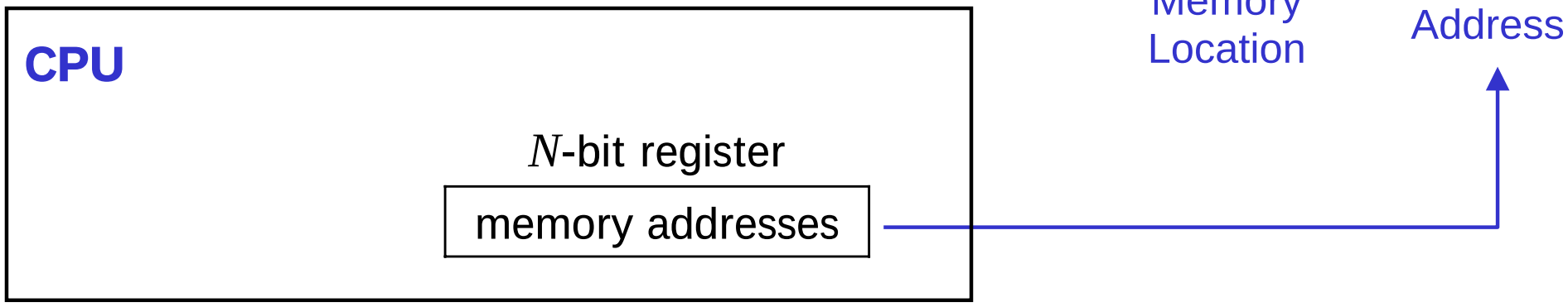
Can form 2^N addresses, from 0 to $(2^N - 1)$

Every byte in RAM has N-bit address

Processor refers to memory locations by physical RAM addresses

Processor stores memory addresses in N-bit address registers

Memory Location	Address
Data Byte	11111…111
Data Byte	11111…110
Data Byte	11111…101
Data Byte	11111…100
…	…
Data Byte	00000…111
Data Byte	00000…110
Data Byte	00000…101
Data Byte	00000…100
Data Byte	00000…011
Data Byte	00000…010
Data Byte	00000…001
Data Byte	00000…000



Word Organization in Memory

Word order

Little endian

- Least significant byte stored at lower address
- Word is stored "little-end-first"
- Example: 4-byte word **`69 b3 36 7d`** stored as

stored byte					**`69`**	**`b3`**	**`36`**	**`7d`**
address	**`07`**	**`06`**	**`05`**	**`04`**	**`03`**	**`02`**	**`01`**	**`00`**

Big endian

- Most significant byte stored at lower address
- Word is stored "big-end-first"
- Example: 4-byte word **`69 b3 36 7d`** stored as

stored byte					**`7d`**	**`36`**	**`b3`**	**`69`**
address	**`07`**	**`06`**	**`05`**	**`04`**	**`03`**	**`02`**	**`01`**	**`00`**

Alignment

Requirement that address of **`s`**-byte data unit be multiple of **`s`**

Formally — **`address A % s = 0`**

- 8086 requires segments to be aligned on 16-byte boundaries
- IA-32 requires pages to be aligned on 4 KB boundaries

Specifying Operands

Immediate

Constant = IMM = numerical value coded into instruction

Register operands

register name = a CPU storage location

REGS[register name] = data stored in register

REGS[R3] = data stored in register **R3 = 11223340**

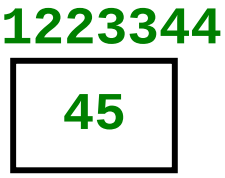


Memory operands

address = a memory storage location

MEM[address] = data stored in memory

MEM[11223344] = data stored at address **11223344** = **45**



Effective Address (EA) — pointer arithmetic

REGS[R3] ← &(variable)

MEM[REGS[R3]+4] = *(&(variable)+4) = *(REGS[R3]+4)

= *(11223340+4) = 45

Structured Operation Models

Defines basic arithmetic procedure and ALU organization

Stack

Push	Pointer ← Pointer – d Stack[Pointer] ← memory/register
Pop	memory/register ← Stack[Pointer] Pointer ← Pointer + d
Binary Op	Stack[Pointer + d] ← Stack[Pointer + d] Op Stack[Pointer] Pointer ← Pointer + d

```
Z = X + Y →     push X
                push Y
                ADD
                pop Z
```

Stack ALU used in Java bytecode

Accumulator

All operations use accumulator **A**

```
Z = X + Y →     load X
                add Y
                store Z
```

Accumulator ALU used in hand calculator

General Register Operation Models

Register-Memory Model

Operands can be stored in any **REGISTER** or **MEMORY** location

```
Z = X + Y →     load R1, X
                add R1, R1, Y
                store Z, R1
```

Register- Register Model

MEMORY operands must be loaded to a **REGISTER**

Also called **LOAD-STORE MODEL**

```
Z = X + Y →     load R1, X
                load R2, Y
                add R1, R1, R2
                store Z, R1
```

Easier to implement

Statistically, most loaded operands are used more than once

Typical Addressing Modes

Mode	Syntax	Memory Access	Use
Register	`R3`	`Regs[R3]`	Register data
Immediate	`#3`	`3`	Constant
Direct (absolute)	`(1001)`	`Mem[1001]`	Static data
Register deferred	`(R1)`	`Mem[Regs[R1]]`	Pointer
Displacement	`100(R1)`	`Mem[100+Regs[R1]]`	Local variable
Indexed	`(R1 + R2)`	`Mem[Regs[R1]+Regs[R2]]`	Array addressing
Memory indirect	`@(R3)`	`Mem[Mem[Regs[R3]]]`	Pointer to pointer
Auto Increment	`(R2)+`	`Mem[Regs[R2]]` `Regs[R2] ← Regs[R2]+d`	Stack access
Auto Decrement	`-(R2)`	`Regs[R2] ← Regs[R2]-d` `Mem[Regs[R2]]`	Stack access
Scaled	`100(R2)[R3]`	`Mem[100+Regs[R2]+Regs[R3]*d]`	Indexing arrays
PC-relative	`(PC)`	`Mem[PC+value]`	Store data relative to program counter (instruction address)
PC-relative deferred	`1001(PC)`	`Mem[PC+Mem[1001]]`	

Typical Operations

Data transfer

Load (r ← m), store (m ← r), move (r/m ← r/m), convert data types

Arithmetic/Logical (ALU)

Integer arithmetic (+ − × ÷ compare shift) and logical (AND, OR, NOR, XOR)

Decimal

Integer arithmetic on decimal numbers

Floating point (FPU)

Floating point arithmetic (+ − × ÷ sqrt trig exp ...)

String

String move, string compare, string search

Control

Conditional and unconditional branch, call/return, trap

Operating System

System calls, virtual memory management instructions

Graphics

Pixel operations, compression/decompression operations

Classic Computer Organization

Considerations in Classic Computer Design

Expensive memory

RAM ~ $5000/MB wholesale in 1977

Poor compilers

Non-optimizing

Bad error messaging

Fast code written or optimized in assembly language

Semantic Gap Argument

Belief among theoreticians in 1960s and 1970

Computer language should imitate natural language

- Large vocabulary
- High redundancy

Implications for Machine Language

Machine Language should be high level

Language defines many instructions

Each instruction performs a lot of work

Language defines many addressing modes

Advantages

Assembly language programming is easier

Each stored instruction in memory more powerful

More power per instruction requires less memory

Classic Machine Design

CISC (Complex Instruction Set Computer)

300+ instruction types

15+ addressing modes

10+ data types

Automated procedure handling

Complex machine implementations

CISC

CISC was conventional wisdom in 1960s and 1970s

Mainframes

Large and expensive computers

Owned by big businesses and governments

Manufacturers: IBM, Control Data, Burrows, Honeywell

From 1960s to 1980s, mainframes were CISC machines

Minicomputers

Smaller computers for smaller organizations

Manufacturers: Digital (PDP/VAX), Data General (Eclipse)

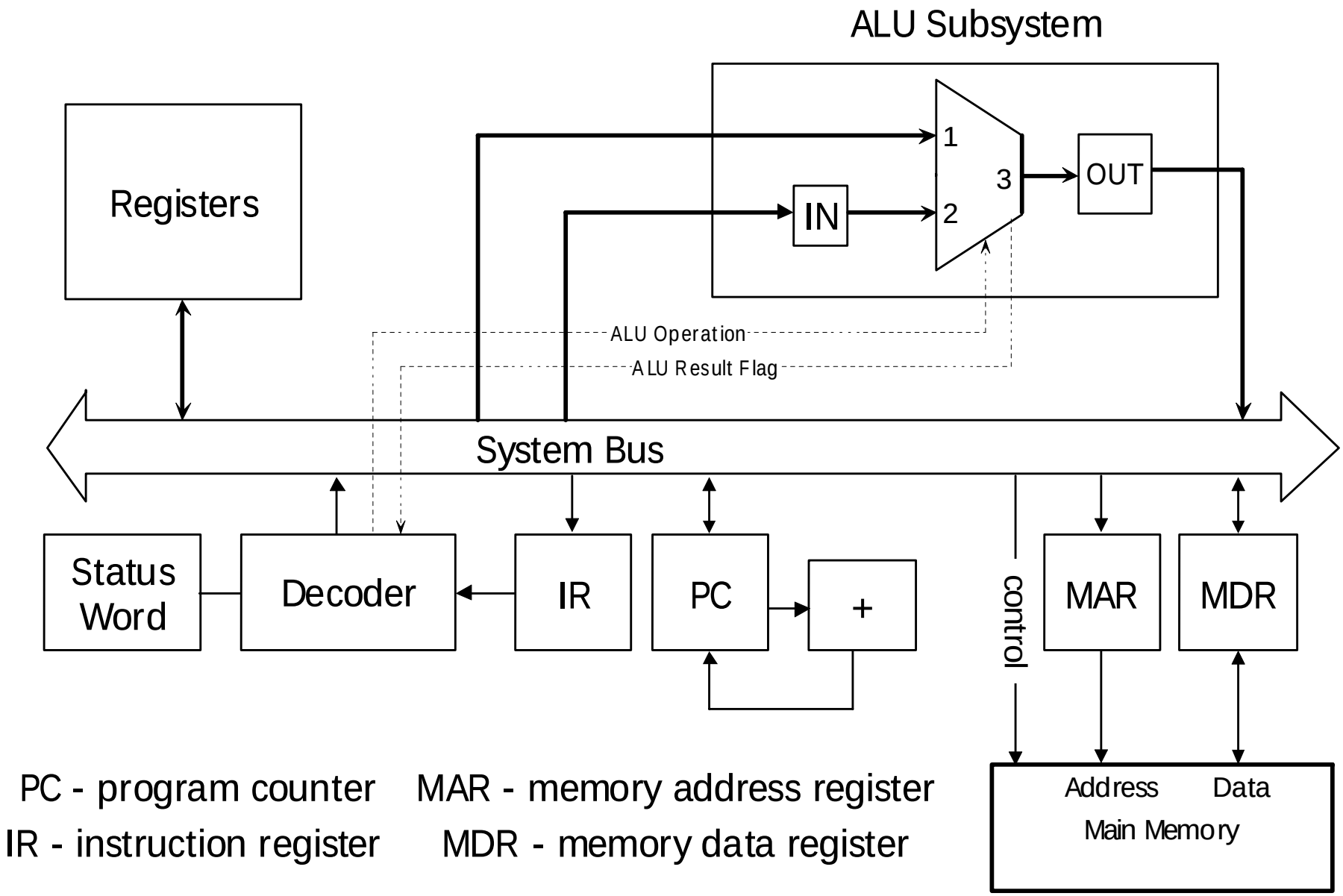
Promoted academic computer science, smaller operating systems (Unix), computer networking

Microcomputers

Intel designed the 8086 (1979) to work like a tiny VAX

The PC is the only CISC computer still manufactured

Physical Implementation



Registers

General Registers

R_0 ... R_{n-1}

Register width is standard integer in ISA

PC

Program Counter

Holds address of next instruction to execute

IR

Instruction Register

Holds binary code of instruction being executed

MAR

Memory Address Register

Holds physical address for RAM access

MDR

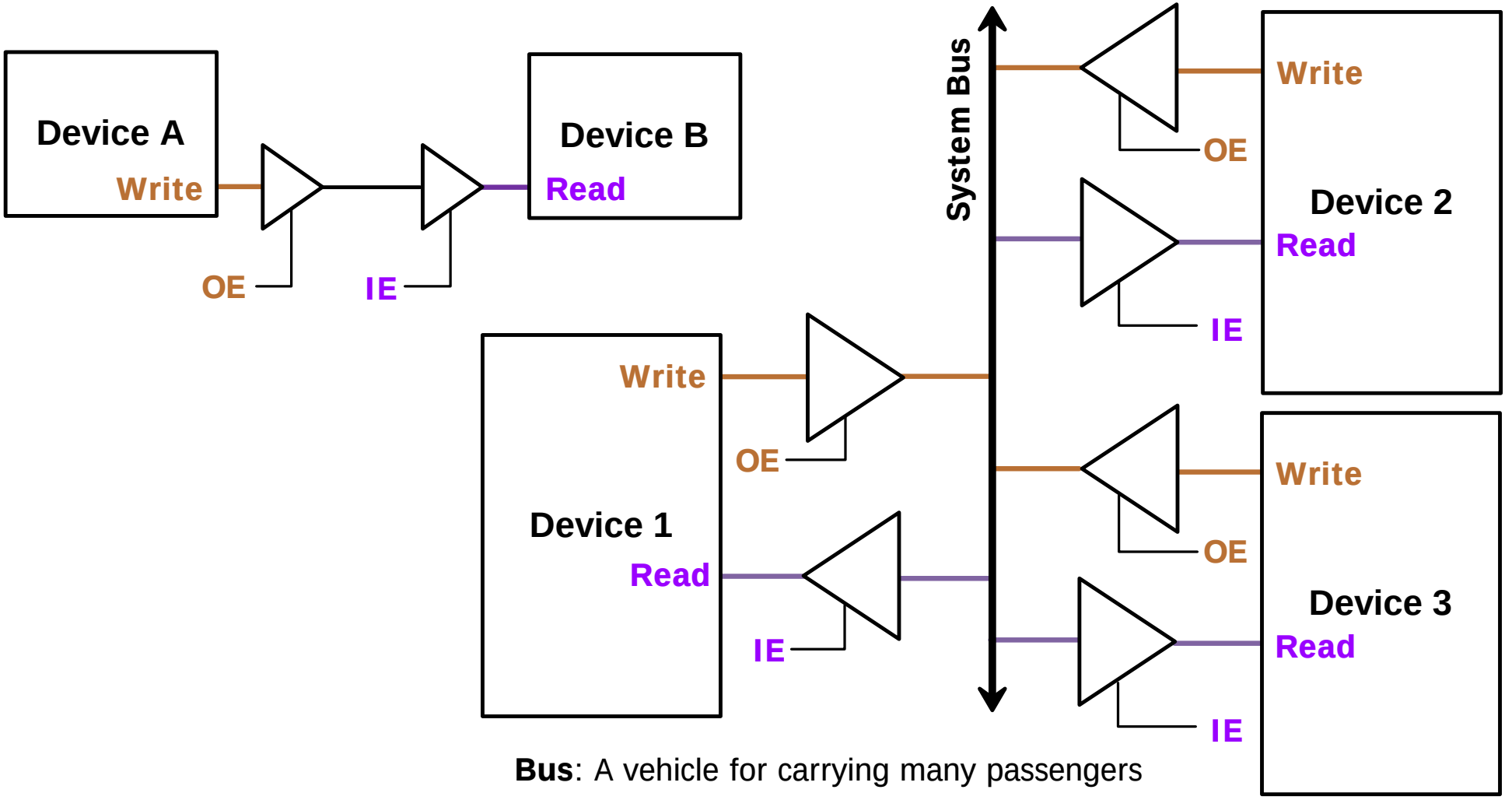
Memory Data Register

Holds data during Read/Write memory operations

Device Communication

A device WRITES with OE = 1 and READS with IE = 1

Von Neumann controller distributes OE and IE signals to devices



Bus: A vehicle for carrying many passengers

Atomic Operations — Instruction Fetch

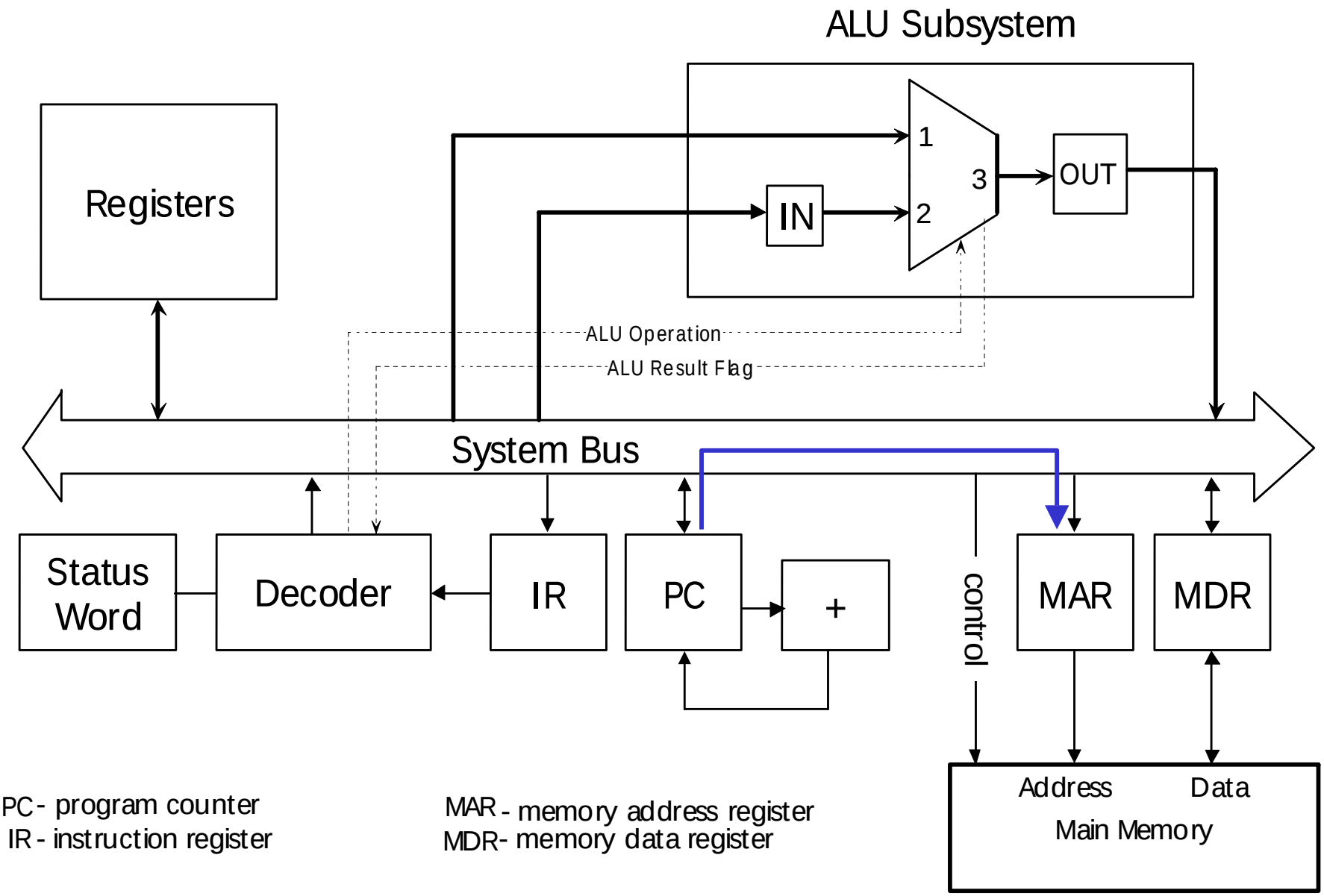
(1) MAR ← PC

(2) READ

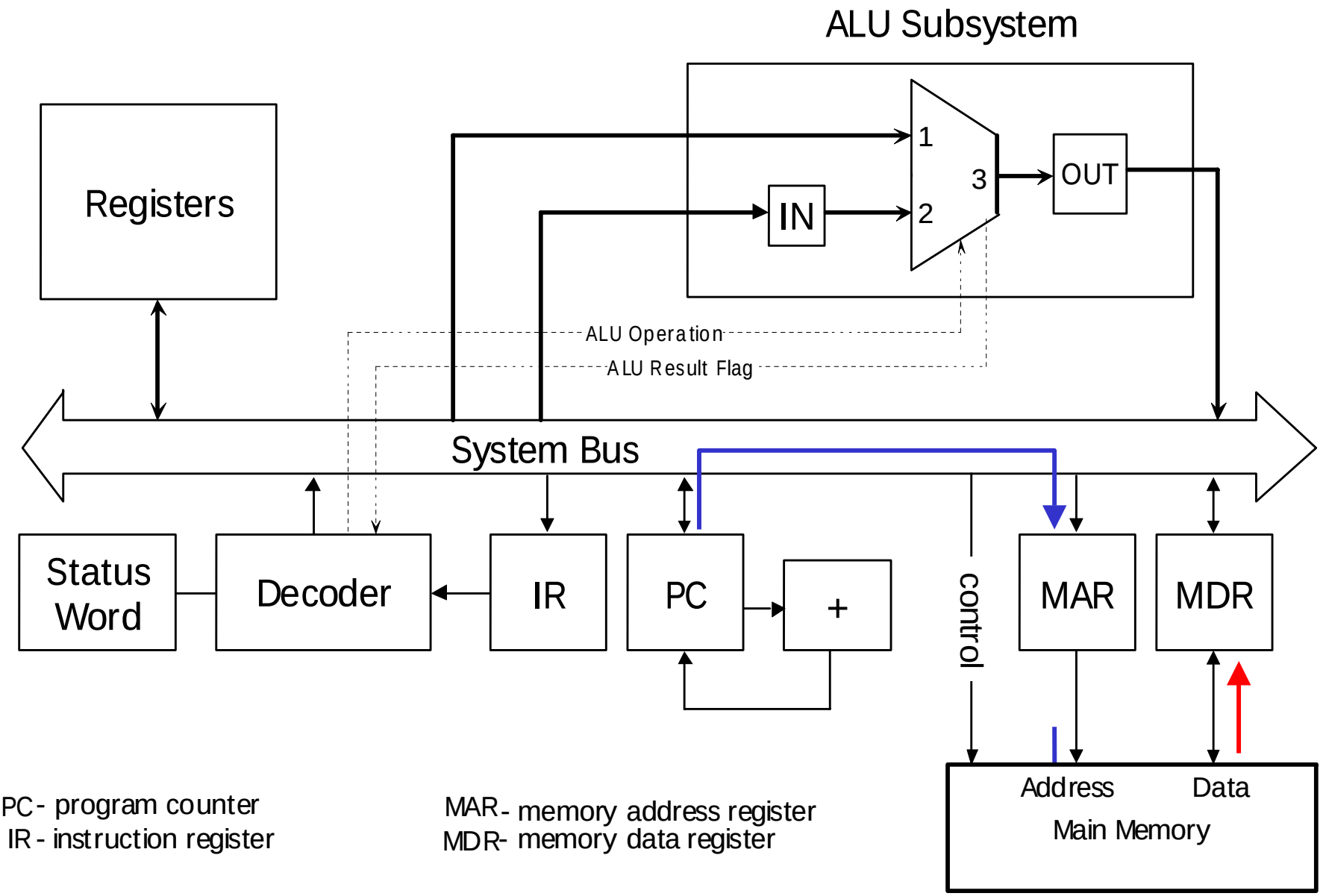
(3) IR ← MDR

(4) PC ← PC + length(instruction)

(1) MAR ← PC


ALU Subsystem
Registers
1
2
3
IN
OUT
ALU Operation
ALU Result Flag
System Bus
Status Word
Decoder
IR
PC
+
control
MAR
MDR
Address
Data
Main Memory
PC - program counter
IR - instruction register
MAR - memory address register
MDR - memory data register

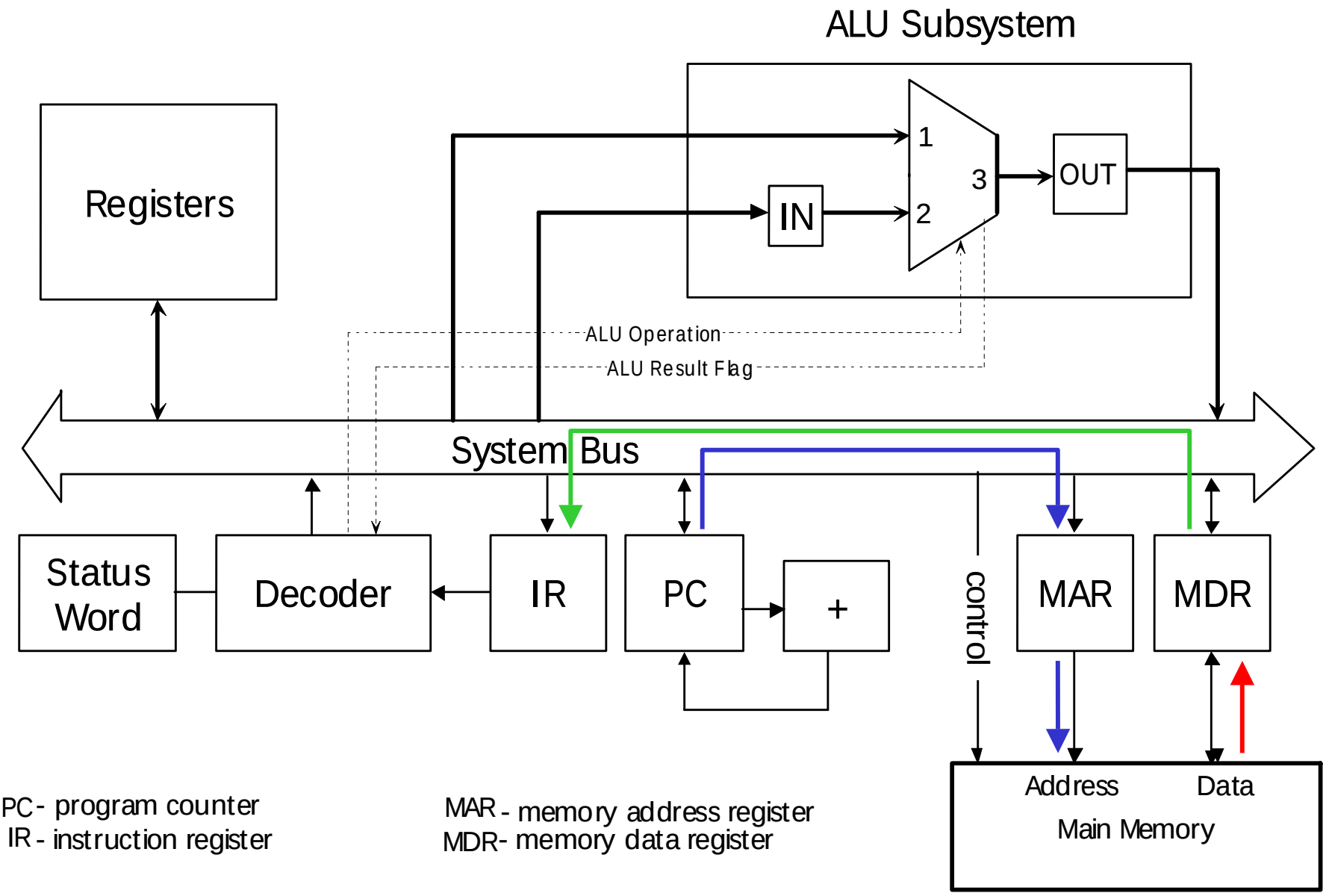
(2) READ



PC - program counter
IR - instruction register

MAR - memory address register
MDR - memory data register

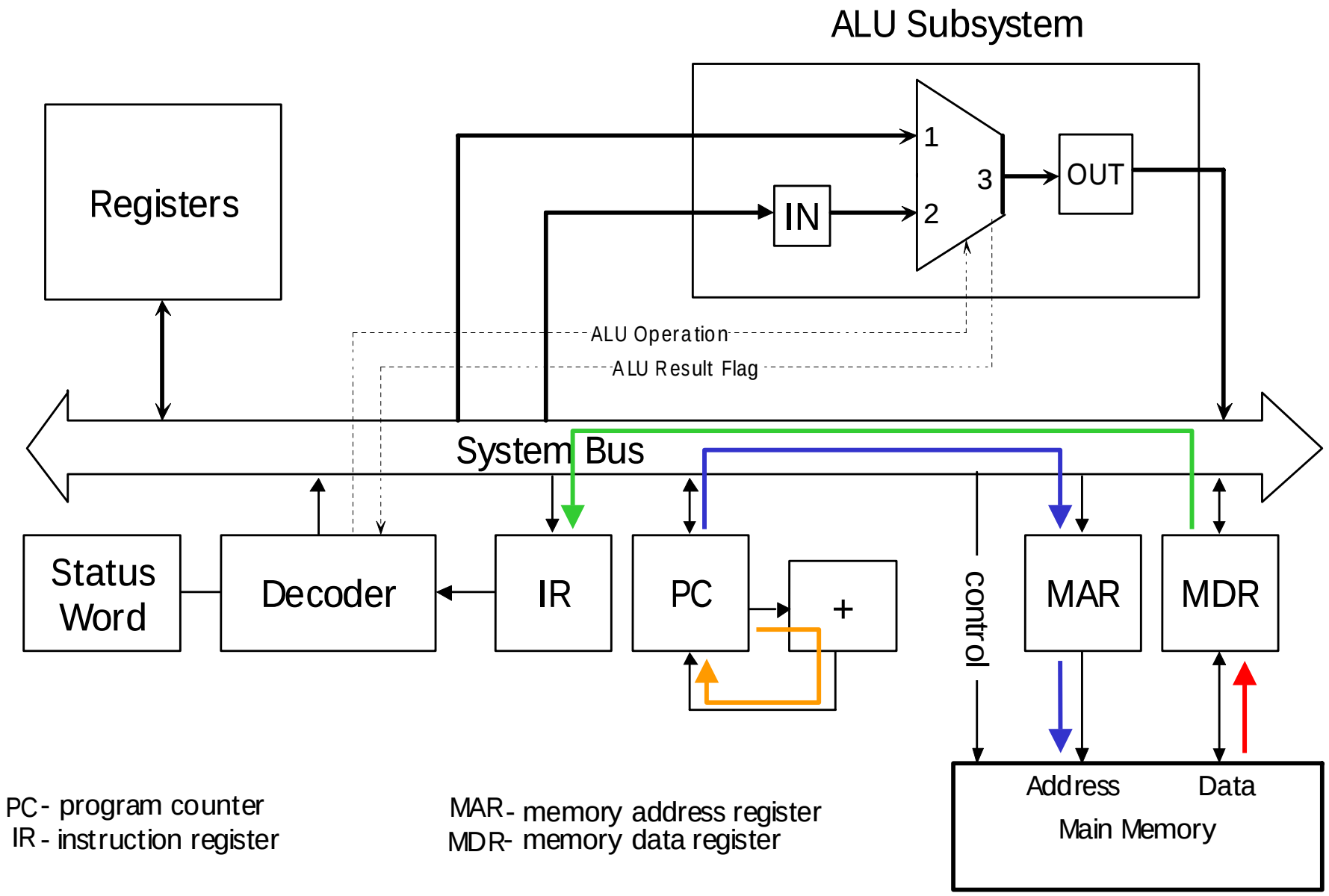
(3) IR ← MDR



PC - program counter
IR - instruction register

MAR - memory address register
MDR - memory data register

(4) PC ← PC + length(instruction)

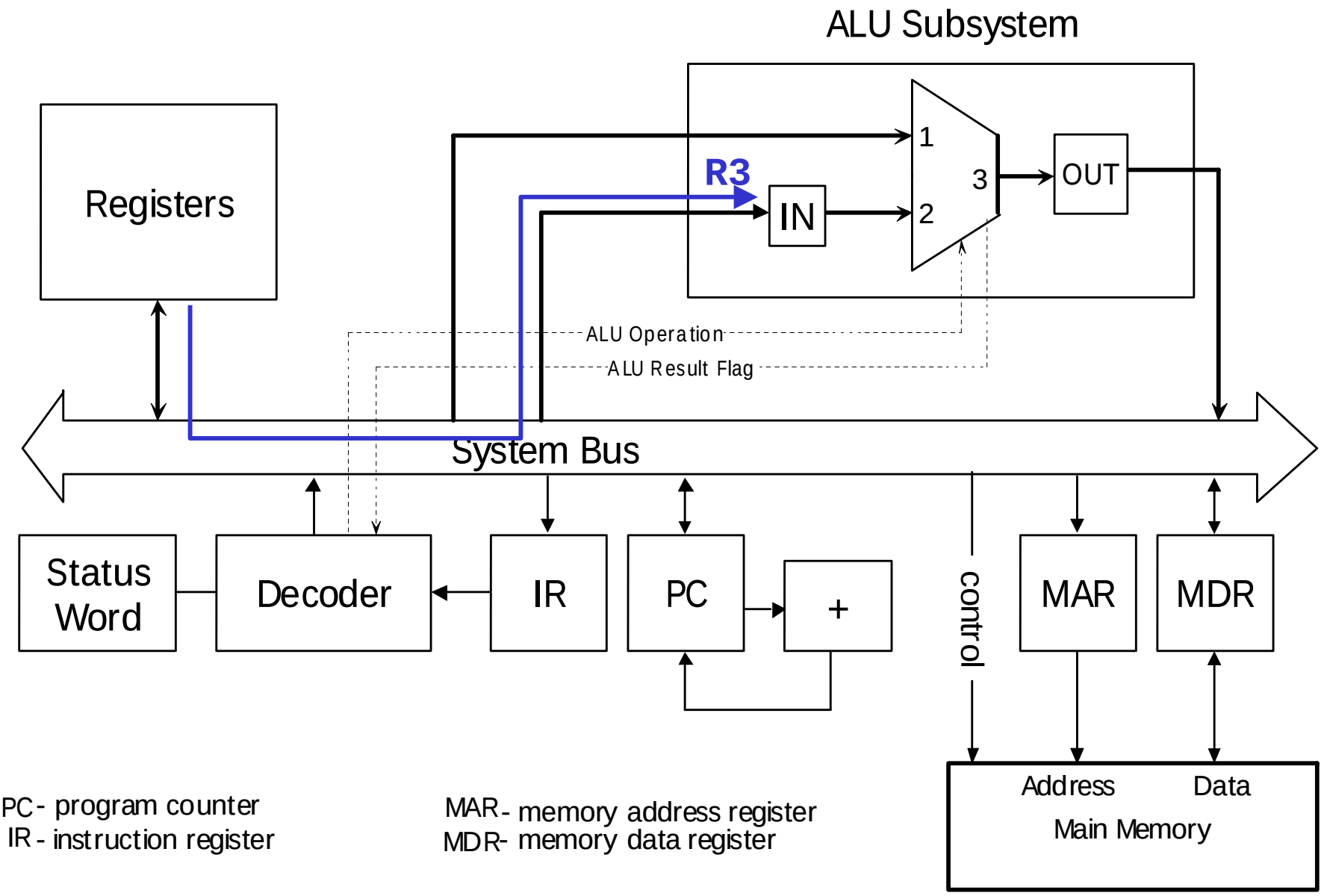


Atomic Operations

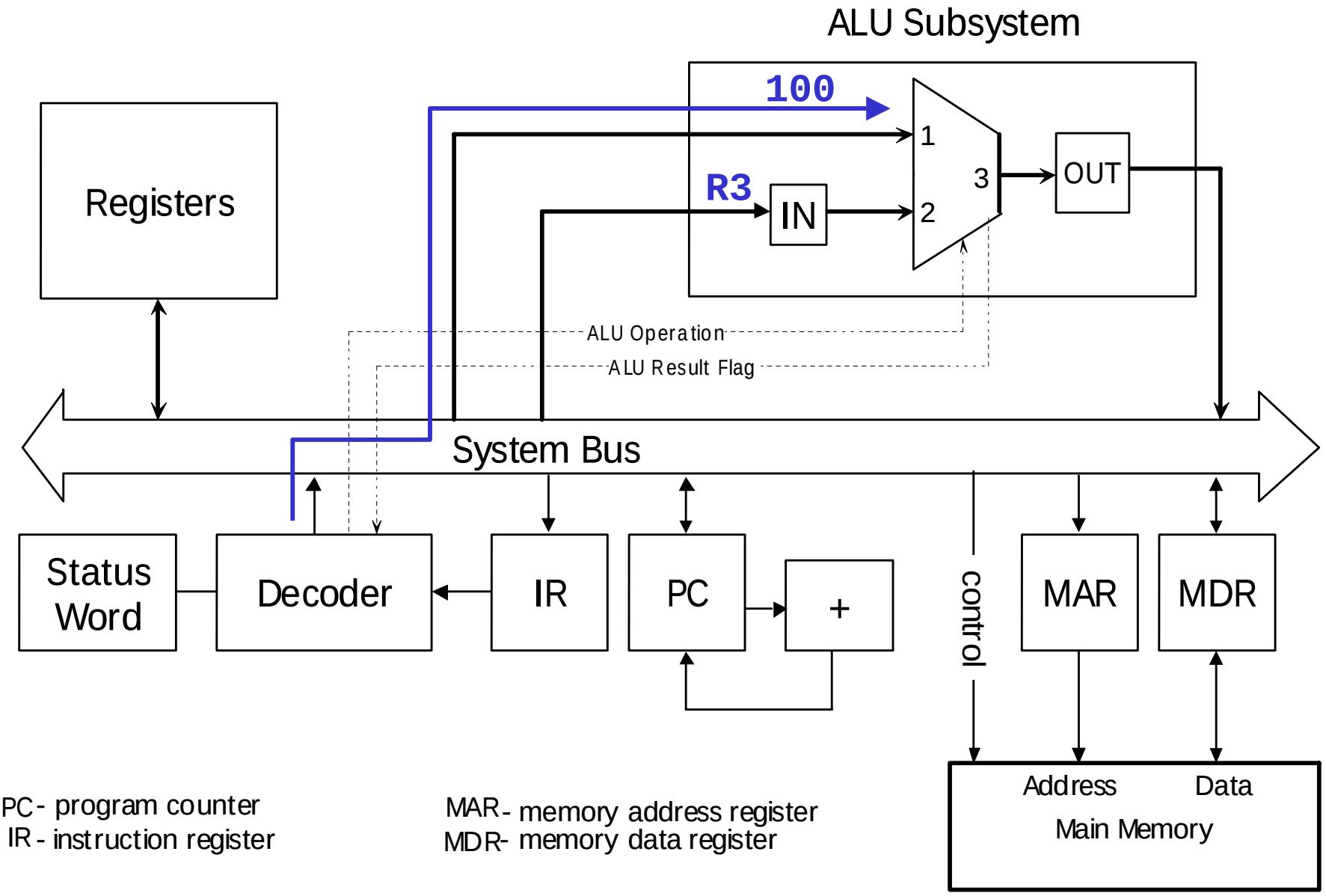
Instruction: `SUB R1, R2, 100(R3)`

```
ALU_IN ← R3
ALU ← 100
ADD
MAR ← OUT
READ
ALU_IN ← MDR
ALU ← R2
SUB
R1 ← OUT
```

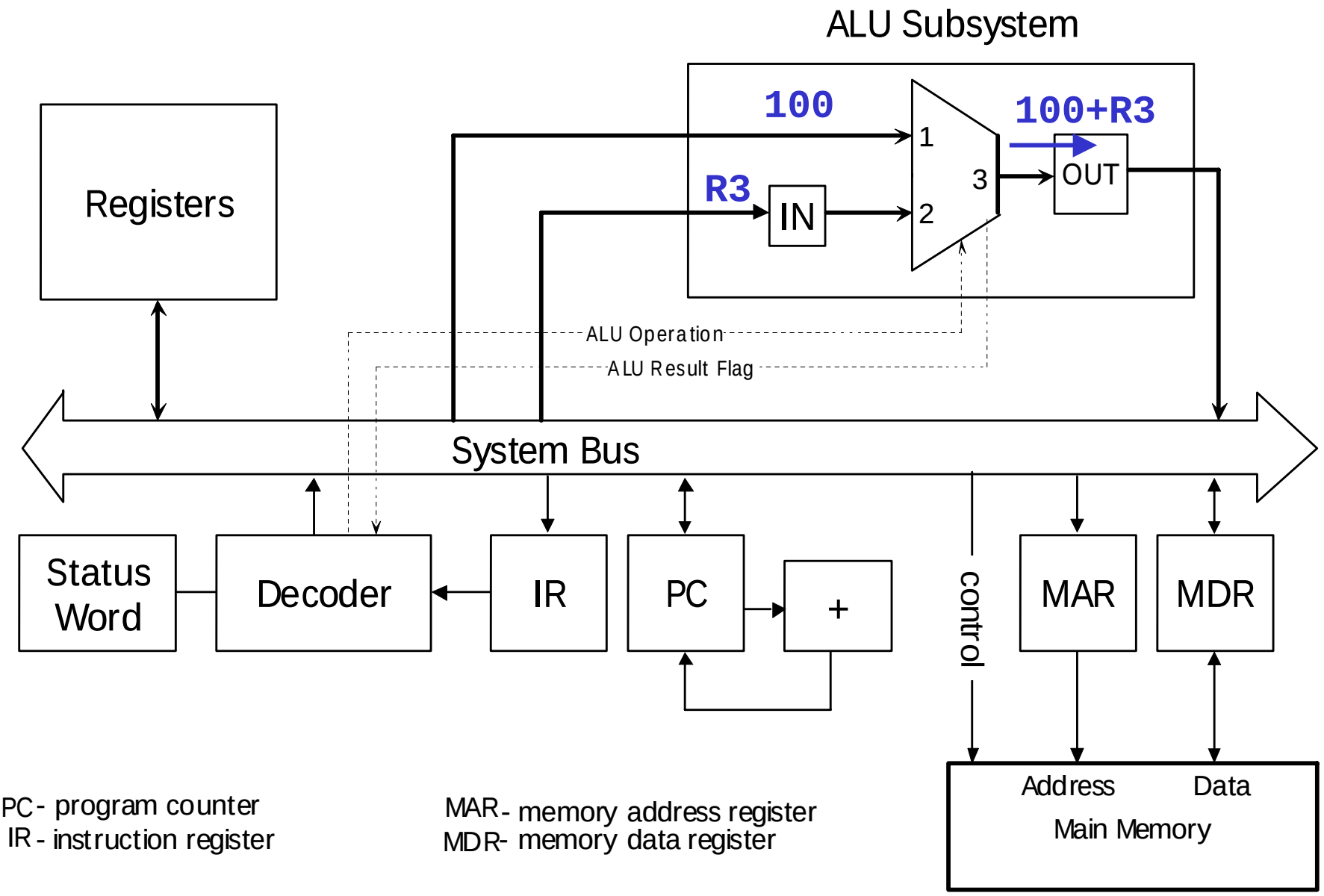
SUB R1, R2, 100(R3): ALU_IN ← R3



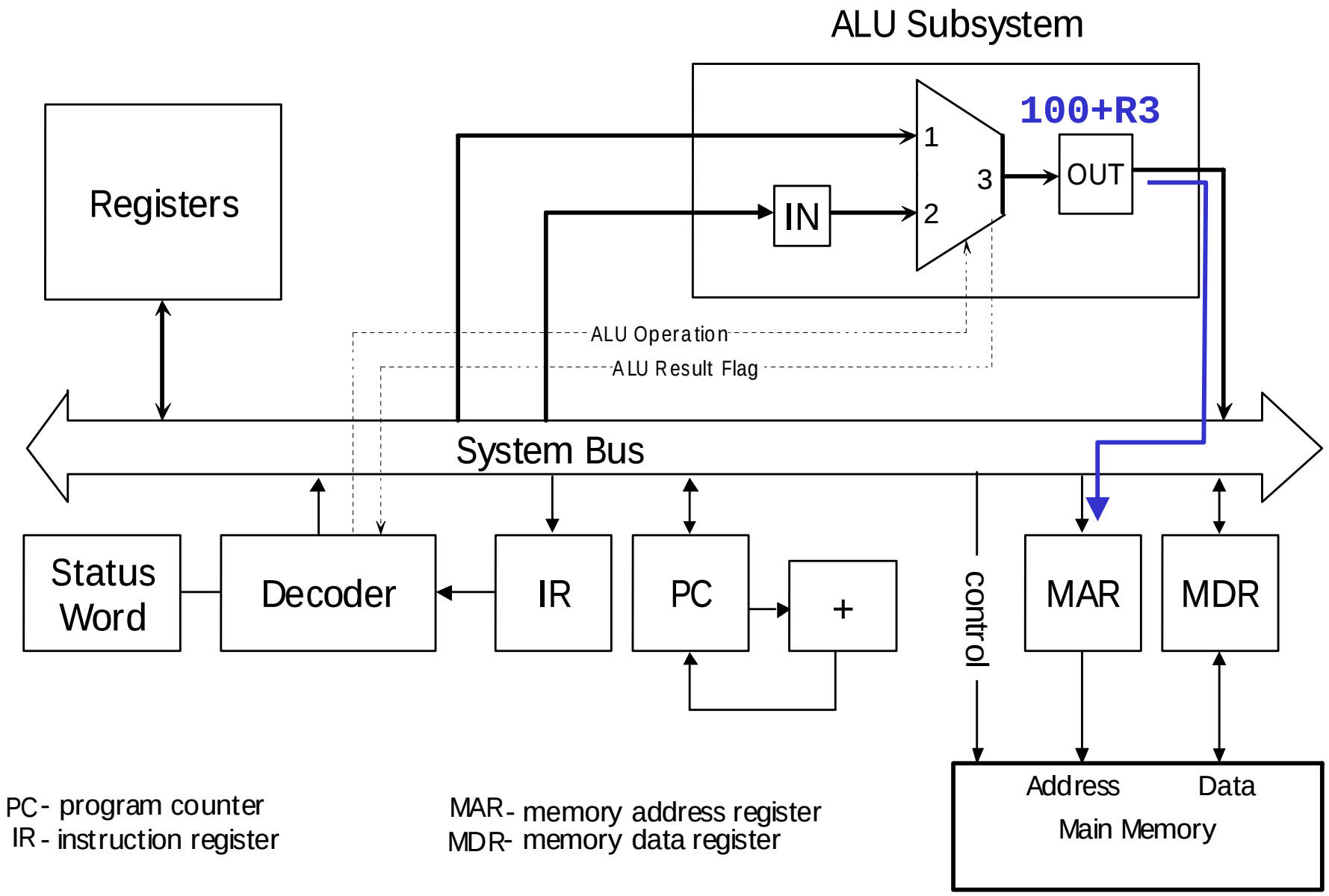
SUB R1, R2, 100(R3): ALU ← 100



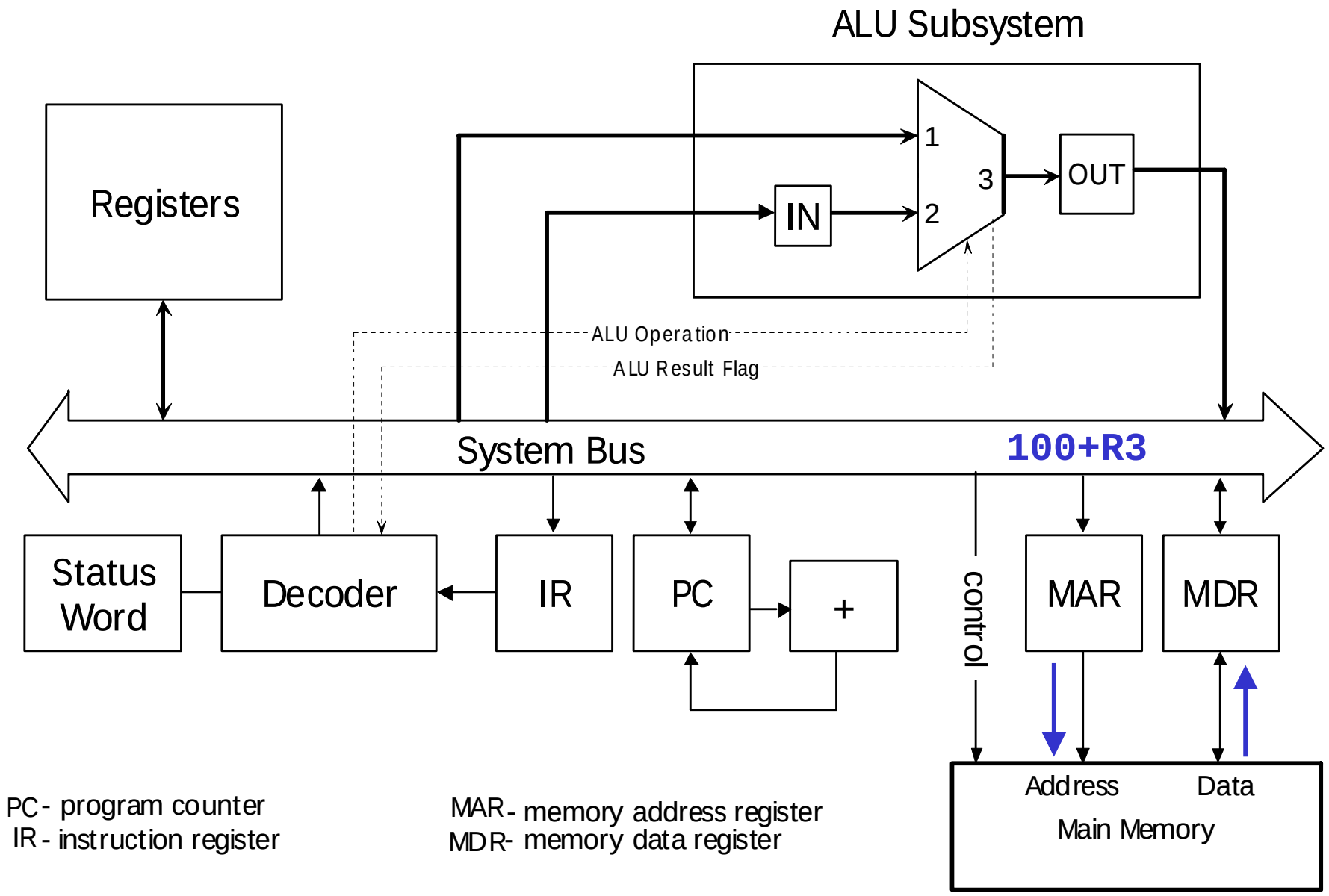
SUB R1, R2, 100(R3): ADD



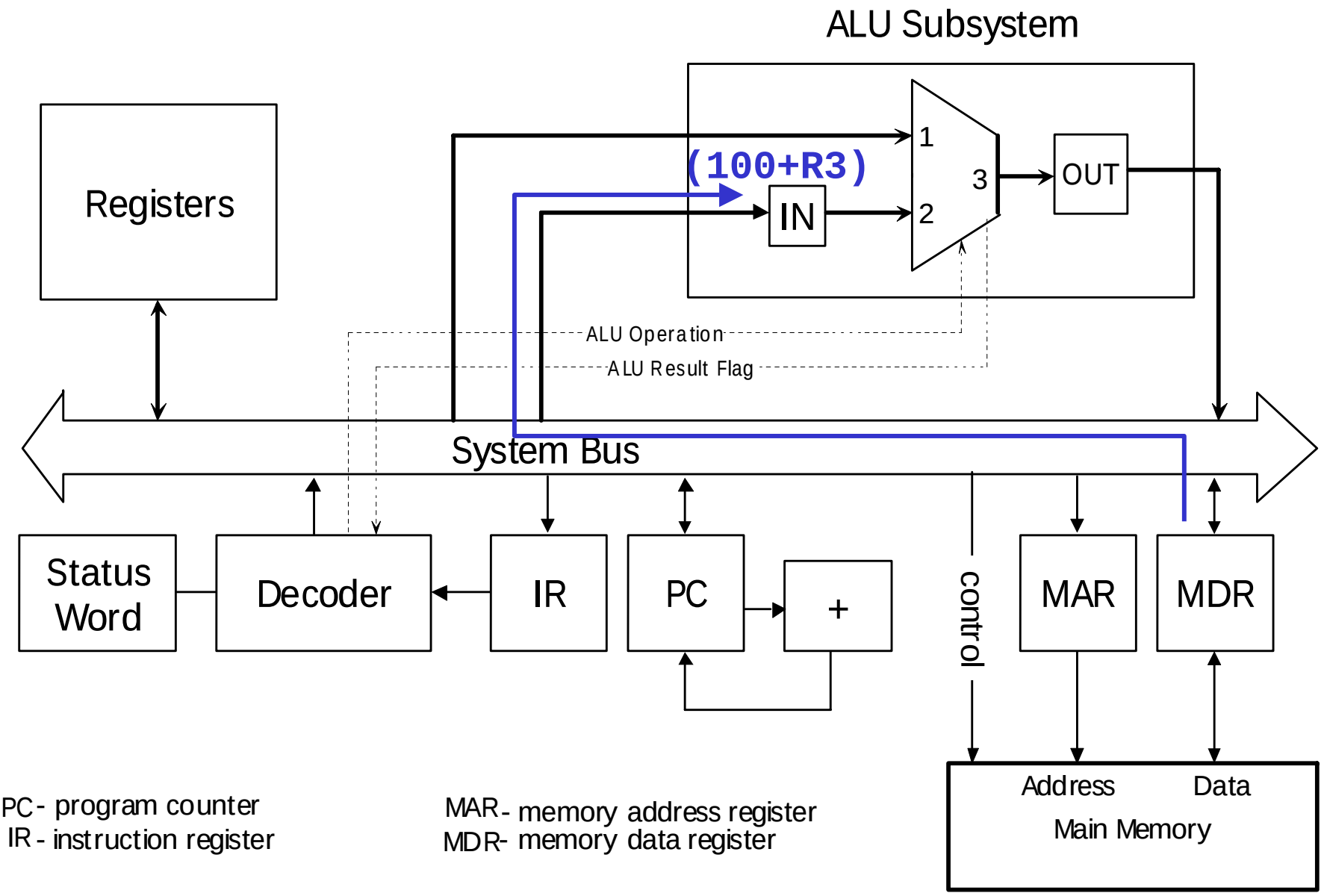
SUB R1, R2, 100(R3): MAR ← OUT



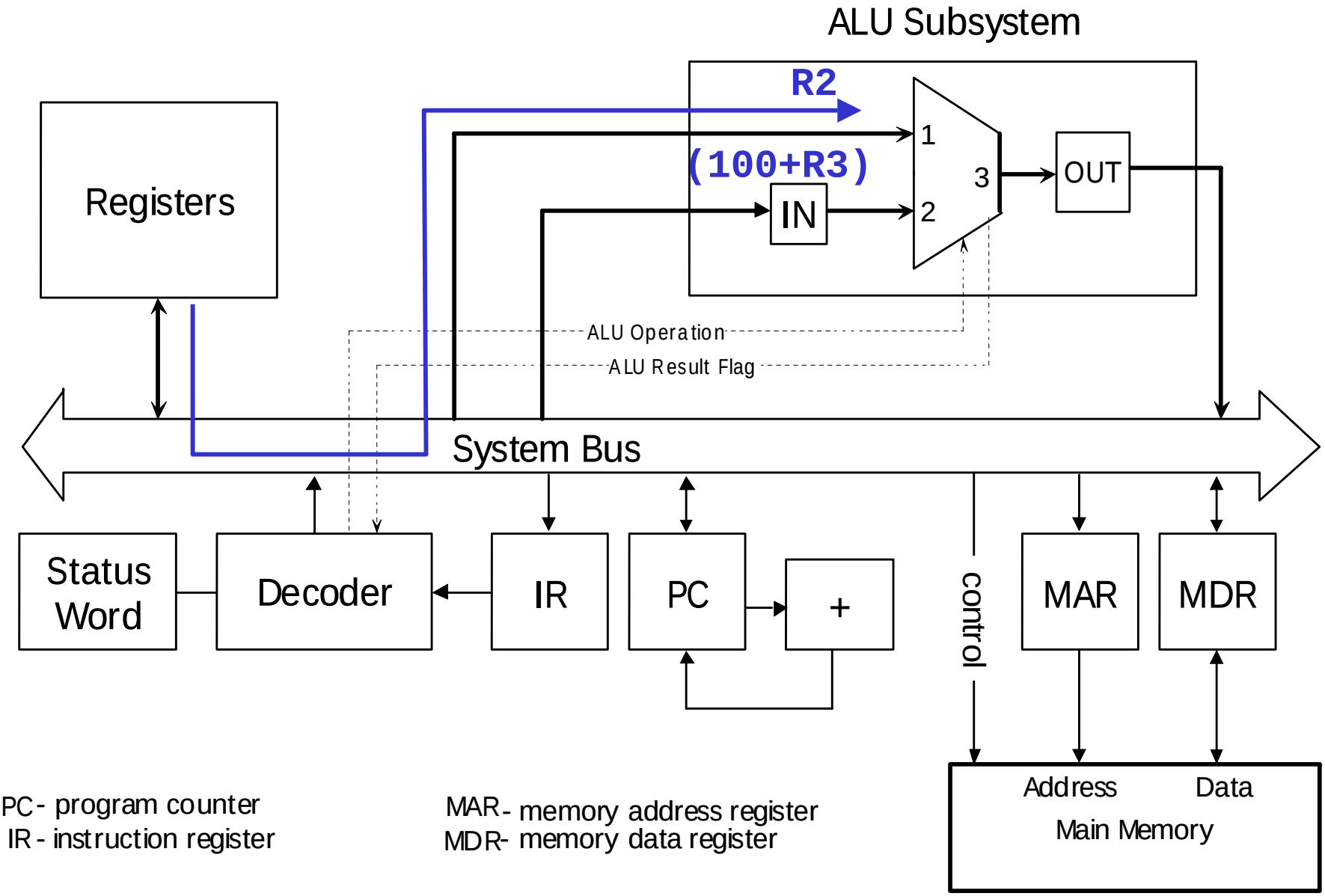
SUB R1, R2, 100(R3): READ



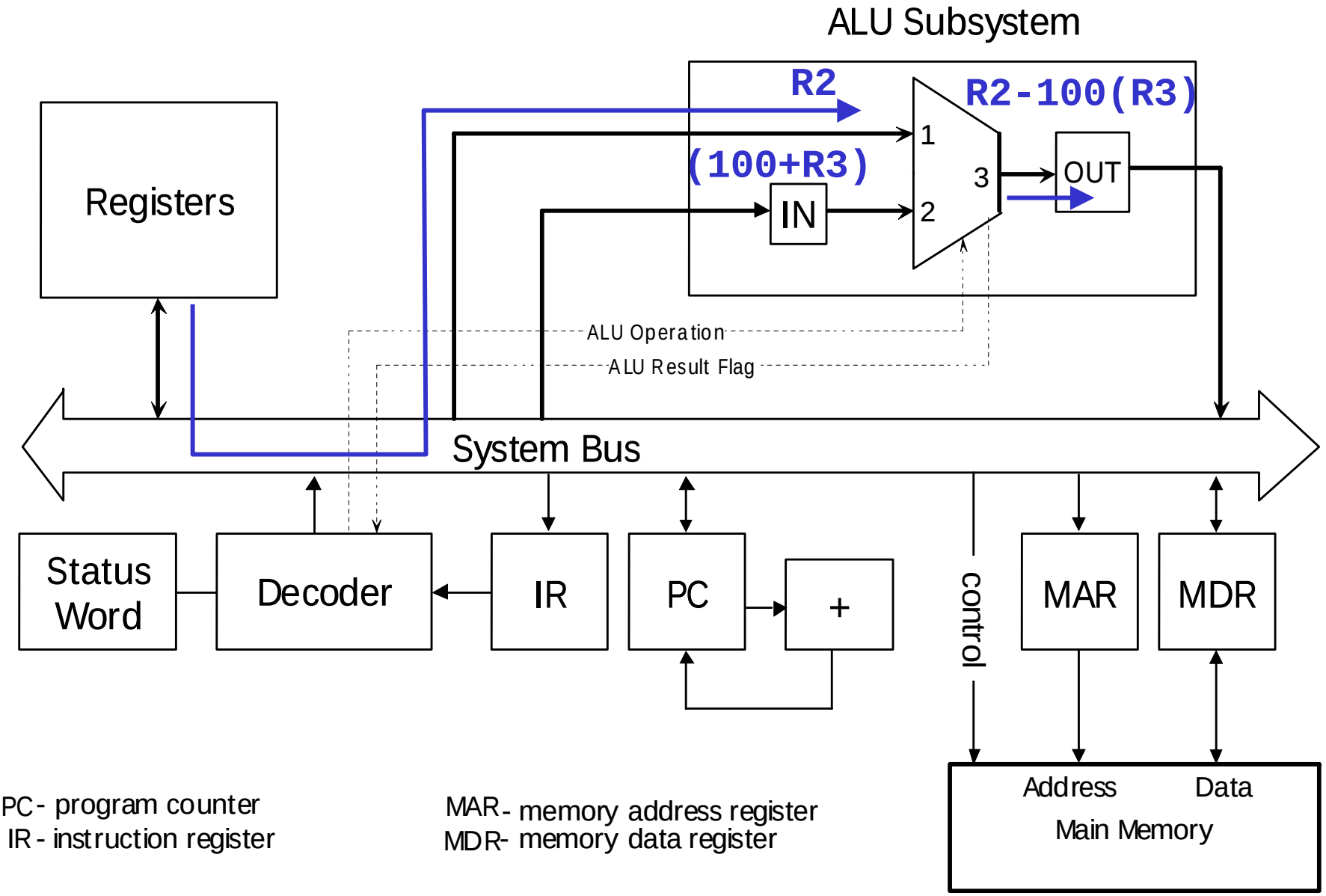
SUB R1, R2, 100(R3): ALU_IN ← MDR



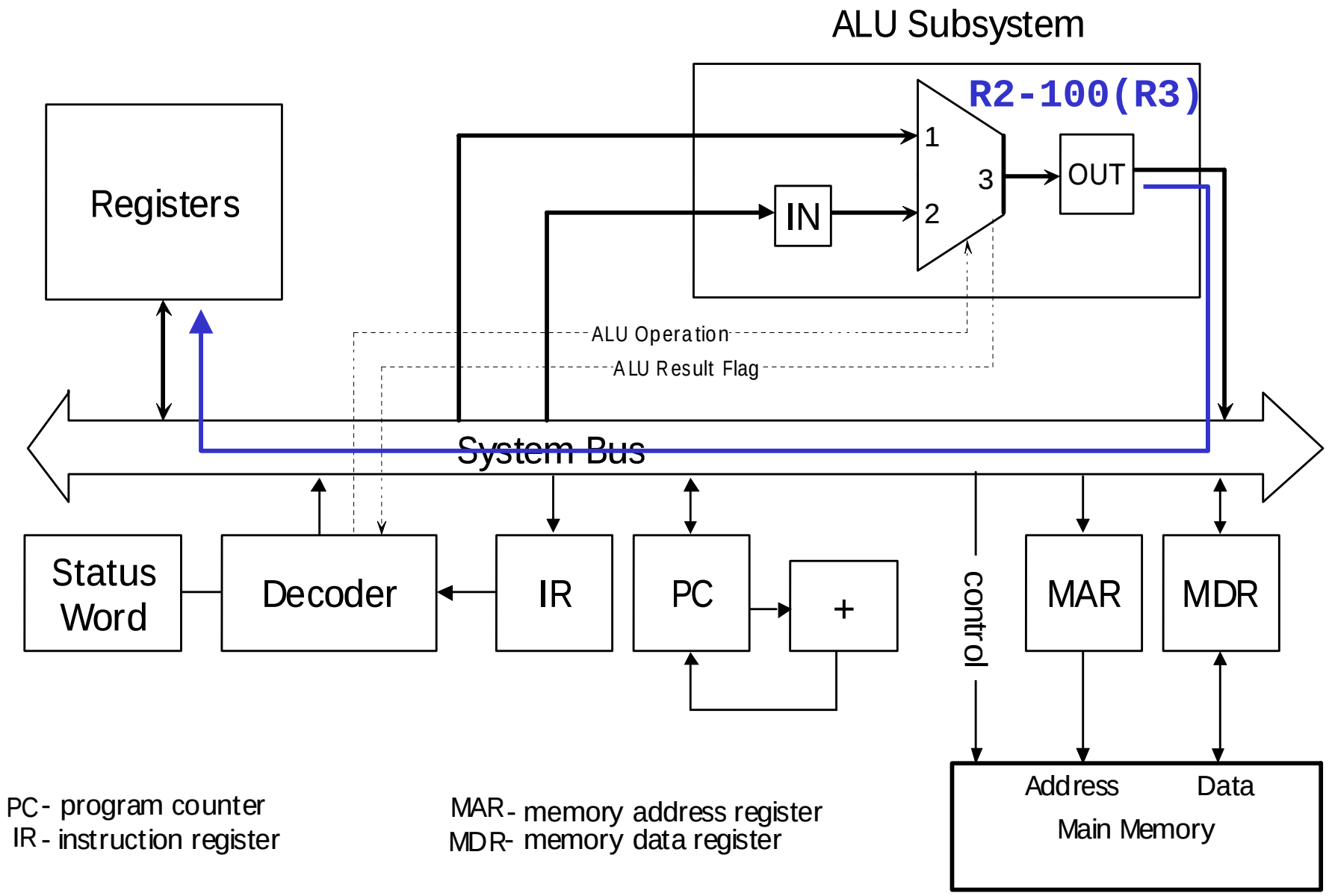
SUB R1, R2, 100(R3): ALU ← R2



SUB R1, R2, 100(R3): SUB



SUB R1, R2, 100(R3): R1 ← OUT



Decoding Machine Instructions

Machine Language Instruction

```
SUB R1, R2, 100(R3)
```

Microcode Instruction Sequence (Microprogram)

```
ALU_IN ← R3
ALU ← 100
ADD
MAR ← OUT
READ
ALU_IN ← MDR
ALU ← R2
SUB
R1 ← OUT
```

Microcode

Microcode

One line of microprogram

Implementation-level atomic operation

Atomic $\Rightarrow$ operation must complete before servicing interrupt

Decoder

"Interprets" machine language instruction into microprogram

Decoder ROM stores microprogram for every legal instruction

New instruction $\Rightarrow$ add microprogram to decoder

Microprogram is sequenced by decoder

State machine for each instruction

Each state provides control signals to every subsystem

Each line of microcode is executed in the correct order

Based on work of Maurice V. Wilkes (1951)

Clock Cycles Per Instruction

Clock Cycle (CC)

Determined by length of longest microcode operations

One line of microcode finishes before next line begins

- Most microcode lines finish in one clock cycle
- Memory access may take several clock cycles

Clock Cycles Per Instruction

Machine language instruction implemented as lines of microcode

Clock Cycles Per Instruction = number of microcode lines

- Memory accesses may take extra clock cycles

Clock cycles for program = number of microcode lines in program

$$CC^{\text{program}} = \sum_{i=\text{instruction types}} \left(\text{Instructions of type } i\right) \times \left(CC^{\text{Instruction of type } i}\right)$$

Instruction type — same basic microcode structure

CISC Creates Anti-CISC Revolution

1974 — 1977

- Data General introduces Eclipse 32-bit CISC minicomputer
- Digital (DEC) introduces VAX 32-bit CISC minicomputer
- First serious inexpensive competition to mainframe computers

1977 — 1990

- Serious computers became available to small organizations
- UNIX developed as minicomputer operating system
- TCP/IP developed to support networks of minicomputers
- Computer Science emerged as separate academic discipline
- Students needed topics for projects, theses, dissertations

1980 — 1990

- Research results on minicomputer performance
  - CISC uses machine resources inefficiently
  - Most machine instructions are rarely used in programs
  - CISC machines run slowly to support unnecessary features