Computer Architecture

Robert Keller

November 2012

Computer Architecture

- A computer is essentially a large collection of **finite-state machines**.
- A common clock is used for all.
- The machines intercommunicate by the output of one machine being the input to another.
- Combinational logic can be interposed between output and input to affect data transformations.

Computer Architecture

- We will work with a simulated tutorial architecture:

  ISC: *Incredibly Simple Computer*

The ISC

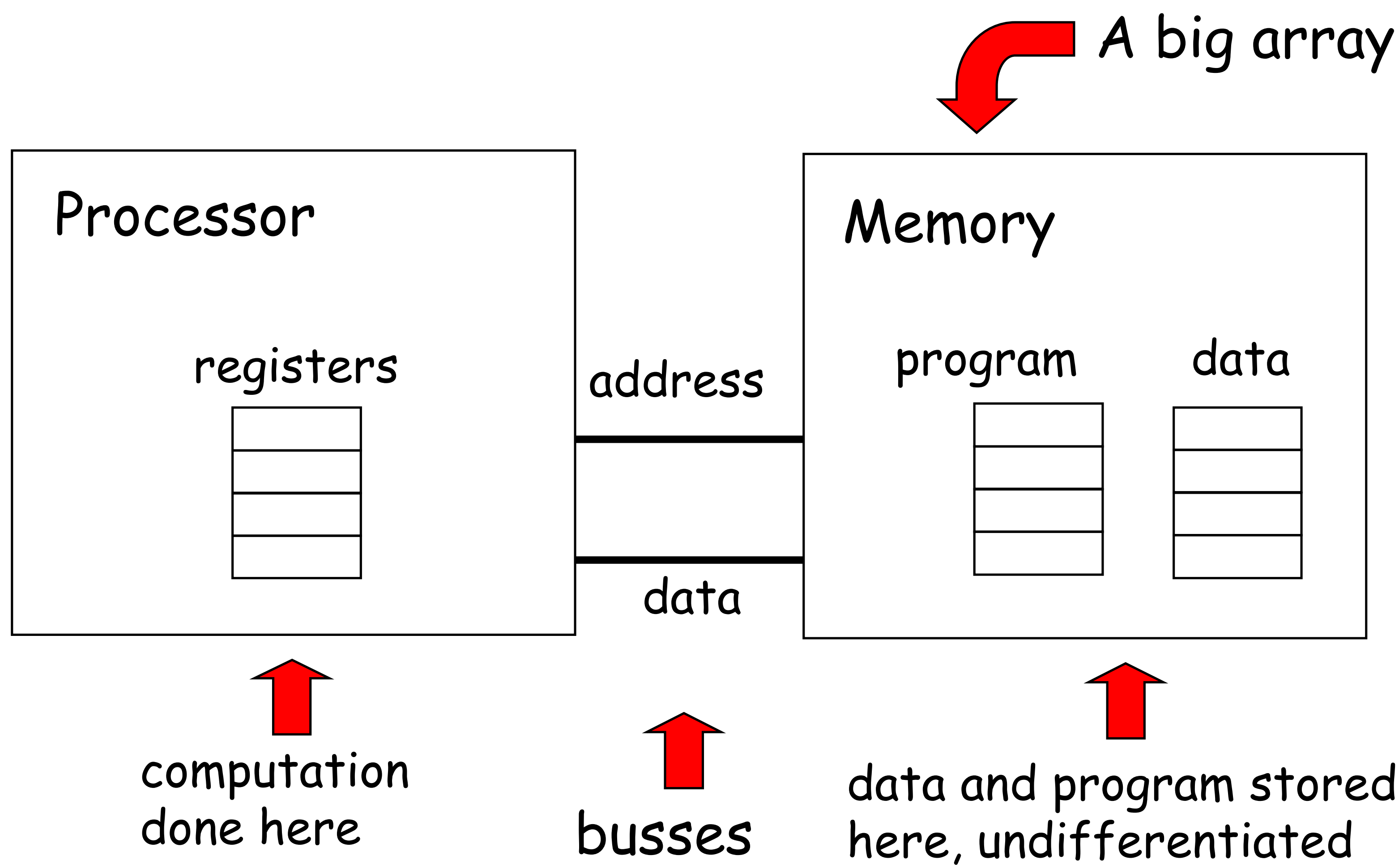
- ISC is an example of a

  RISC (Reduced Instruction Set Computer),

  as opposed to a

  CISC (Complex Instruction Set Computer)

ISC Design


A big array
Processor
registers
address
data
Memory
program
data
computation
done here
busses
data and program stored
here, undifferentiated

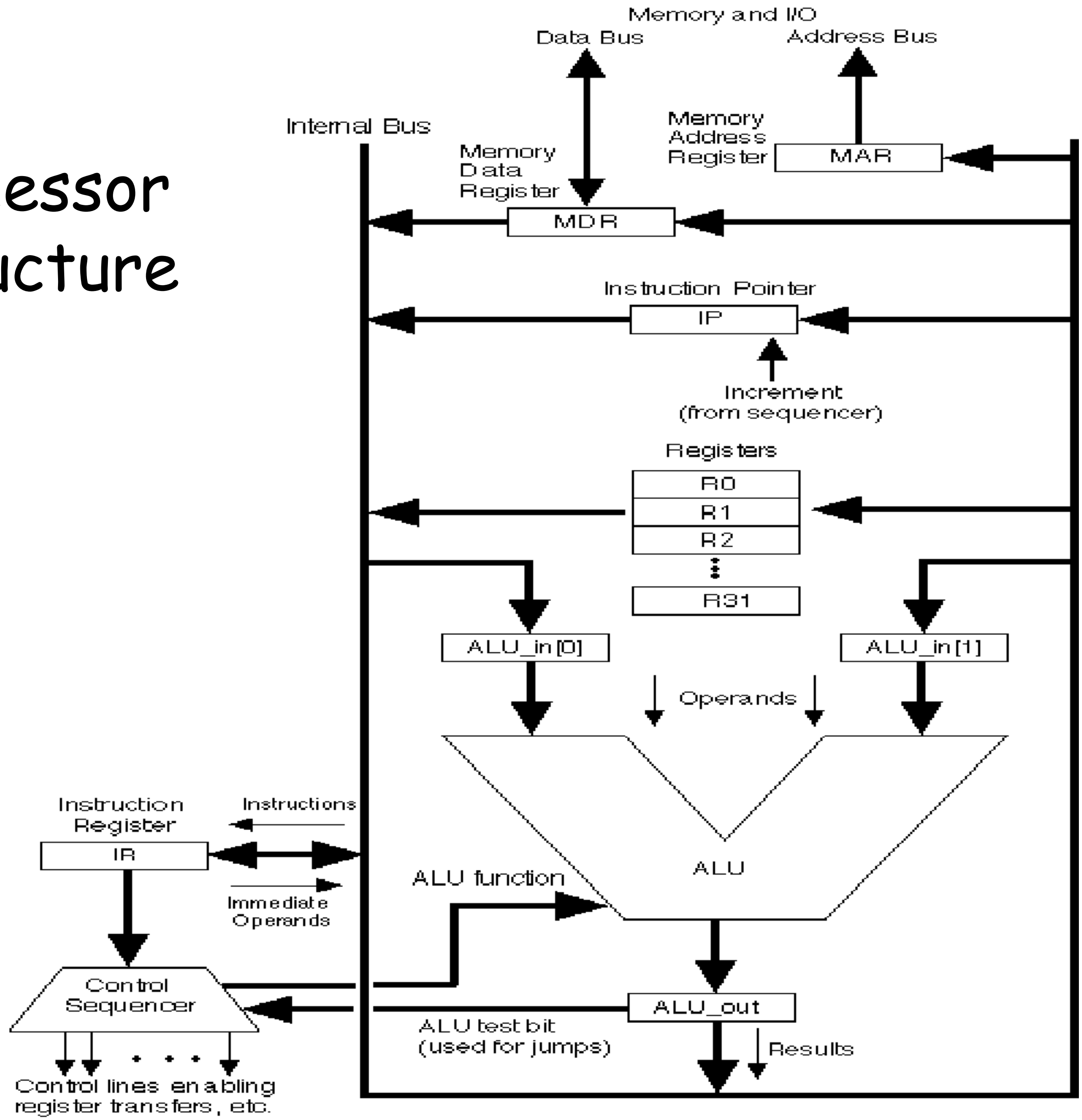
ISC Processor Components (1)

- **General Registers**: Hold operands and results, and addresses for jumps in program.
- **IR** (Instruction Register) holds currently executing instruction.
- **IP** (Instruction Pointer): Holds address of next instruction.
- **MDR** (Memory Data Register) holds data en-route to/from memory.
- **MAR** (Memory Address Register) holds address of memory location for data.

ISC Processor Components (2)

- **ALU** (Arithmetic-Logic Unit) Combinational unit performing addition, subtraction, logical operations, shifting, etc.
- **ALU registers**: Registers holding operands and results for ALU
- **Control sequencer**: Finite-state machine sequencing register and 3-state strobes, based upon the contents of the IR (Instruction Register)

ISC Processor Structure



Instruction Dichotomies

- Instructions that access memory ("memory access" instructions)
- Instructions that include an operand in the instruction itself ("immediate" instructions)
- Instructions that change instruction location ("jump" instructions)

Non-memory access instructions

add *Ra Rb Rc*

register indices (absolute or symbolic)

reg[Ra] = reg[Rb] + reg[Rc];

similarly for:

- sub
- mul
- div
- and
- or
- comp
- shr
- shl

"Immediate" instructions

C is a constant

It can be intended for arithmetic, logic, or as an address.

load immediate:

lim *Ra C*

→ Reg[Ra] = C;

add immediate:

aim *Ra C*

→ Reg[Ra] += C;'

Memory access instructions

load:

load *Ra Rb*

reg[Ra] = mem[reg[Rb]];

The address to be used is in one of the general registers.

store:

store *Ra Rb*

mem[reg[Ra]] = reg[Rb];

Jump instructions

jump-if-equal:

jeq *Ra Rb Rc*

Jump to address in Ra

if reg[Rb] == reg[Rc]

The address to be jumped to is in one of the general registers.

Similarly for:

jne jlt
jgt jlte
jgte

Jump instructions

The address to be jumped to is in one of the general registers.

jump-unconditionally

junc *Ra*

Jump to address in Ra.

Jump instructions

The address to be jumped to is in one of the general registers.

jump-to-subroutine

jsub *Ra Rb*

Jump to address in Ra.

The next location after the current one is put in Rb.

Control Sequencer FSM

- Inputs are bits from instruction being interpreted
- Outputs are strobes to various registers, 3-state devices, etc.

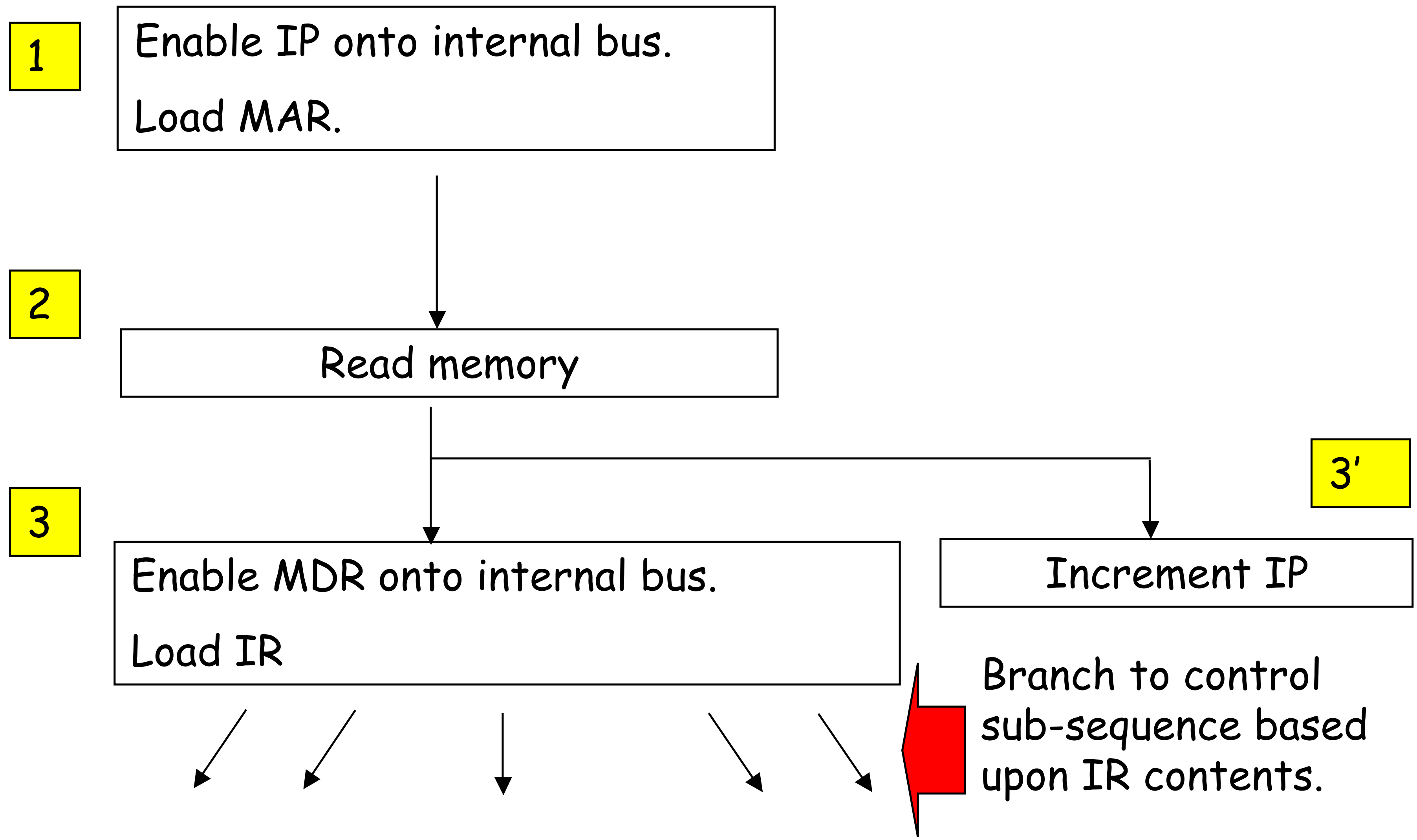
Control Sequence

- There is a sequence common to all instructions in which the instruction is fetched from memory,

  followed by

- A sequence particular to the type of instruction being executed.

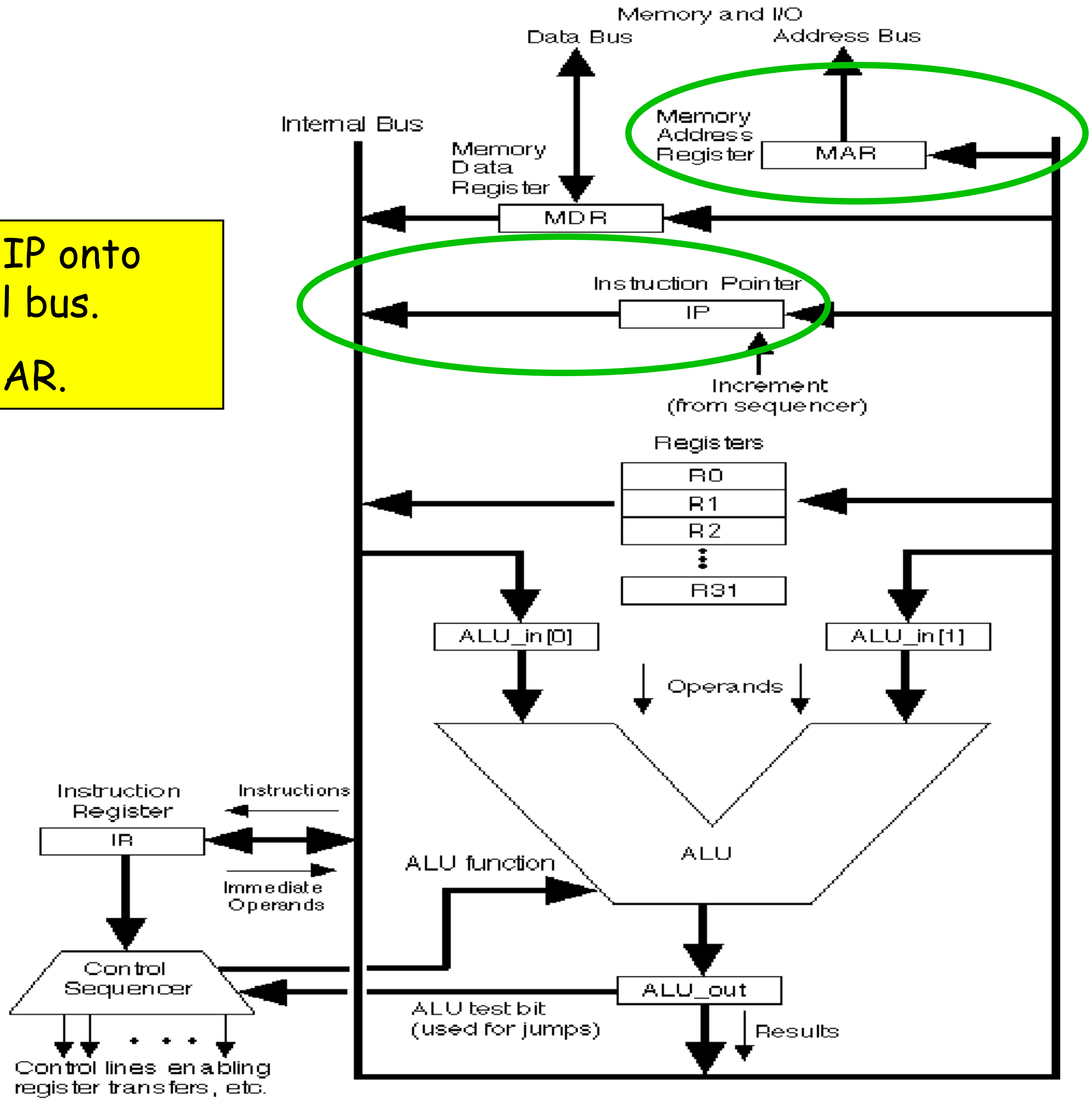
ISC Instruction Fetch



1

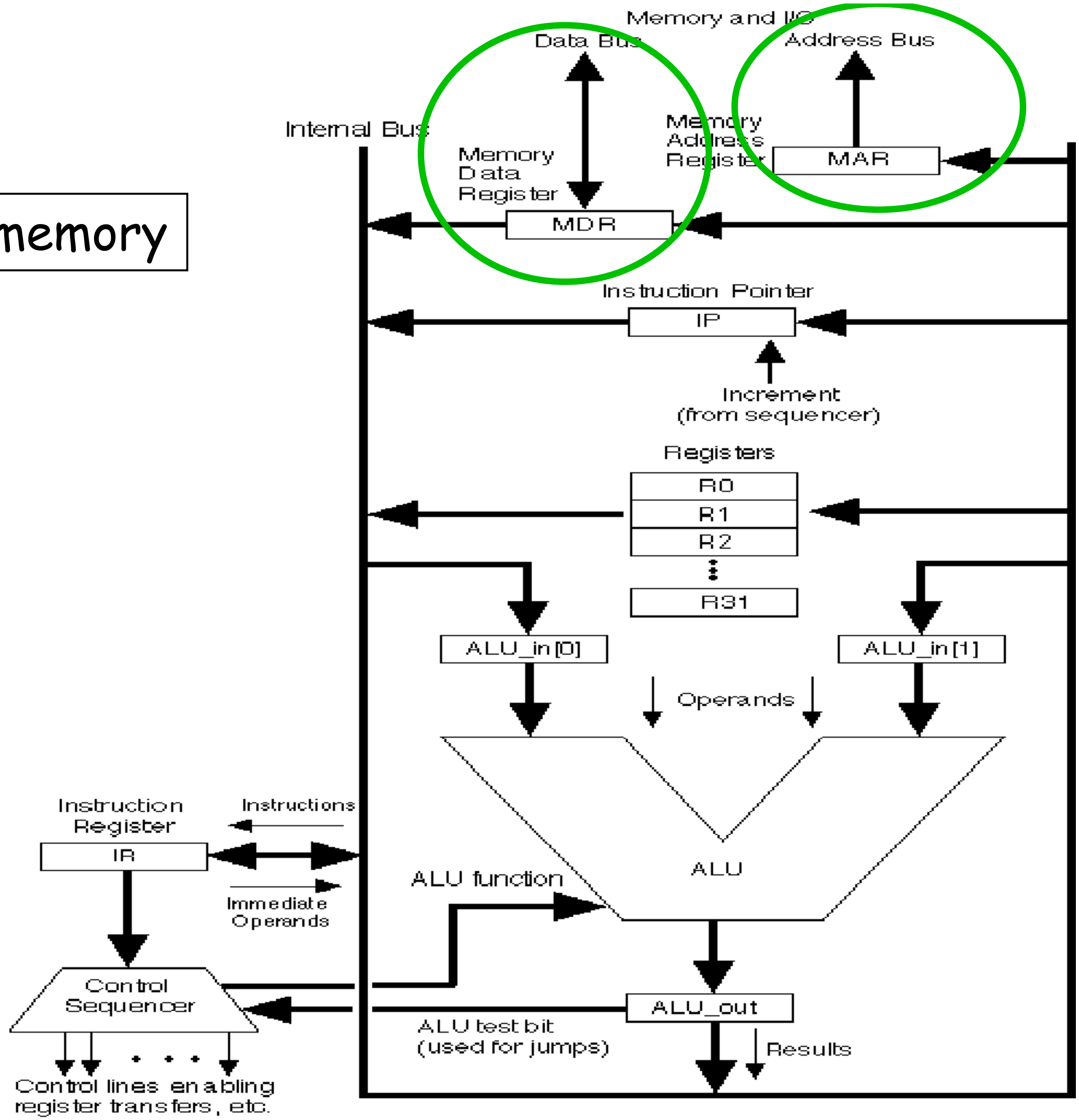
Enable IP onto internal bus.

Load MAR.



2

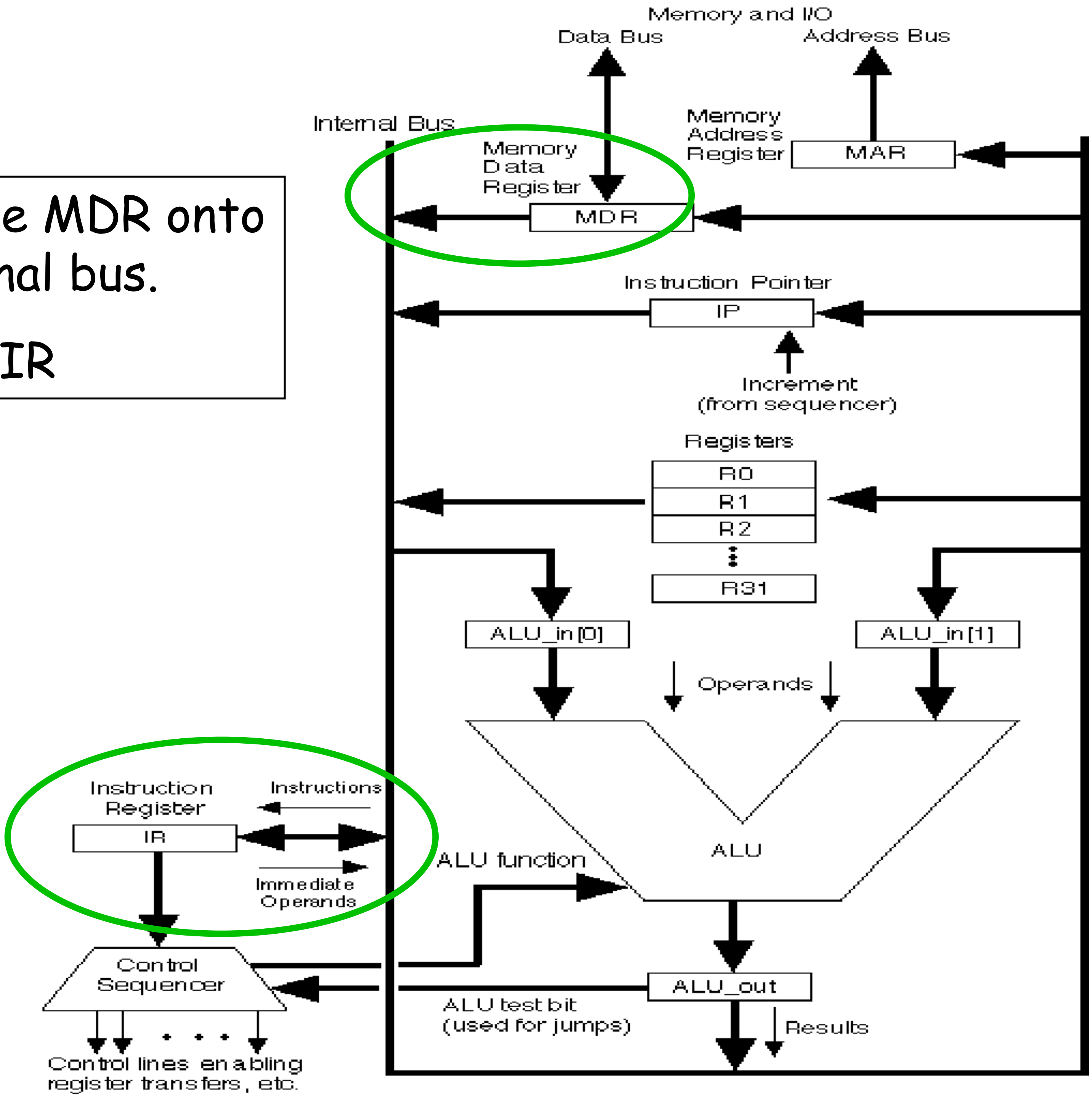
Read memory



3

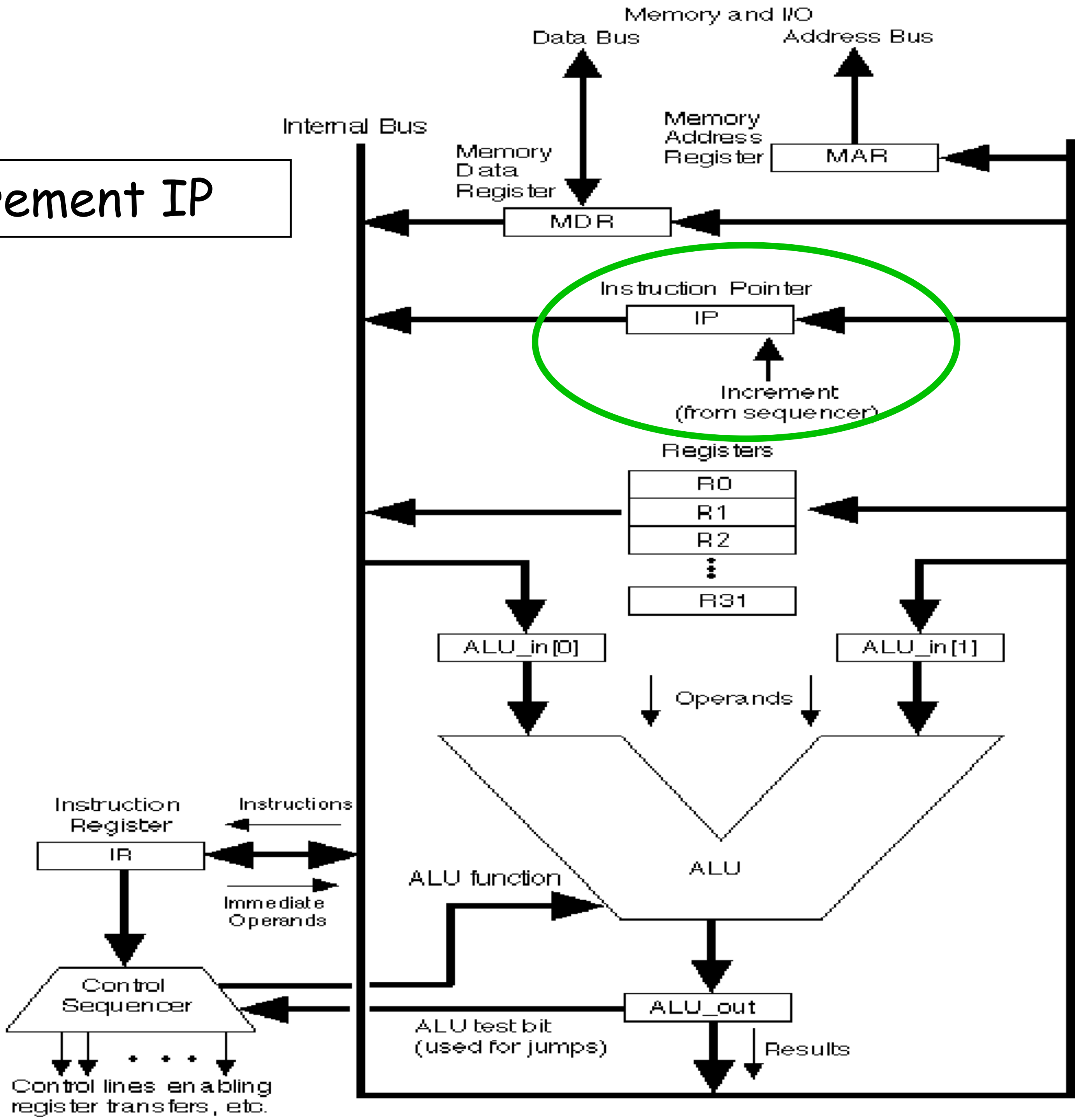
Enable MDR onto internal bus.

Load IR

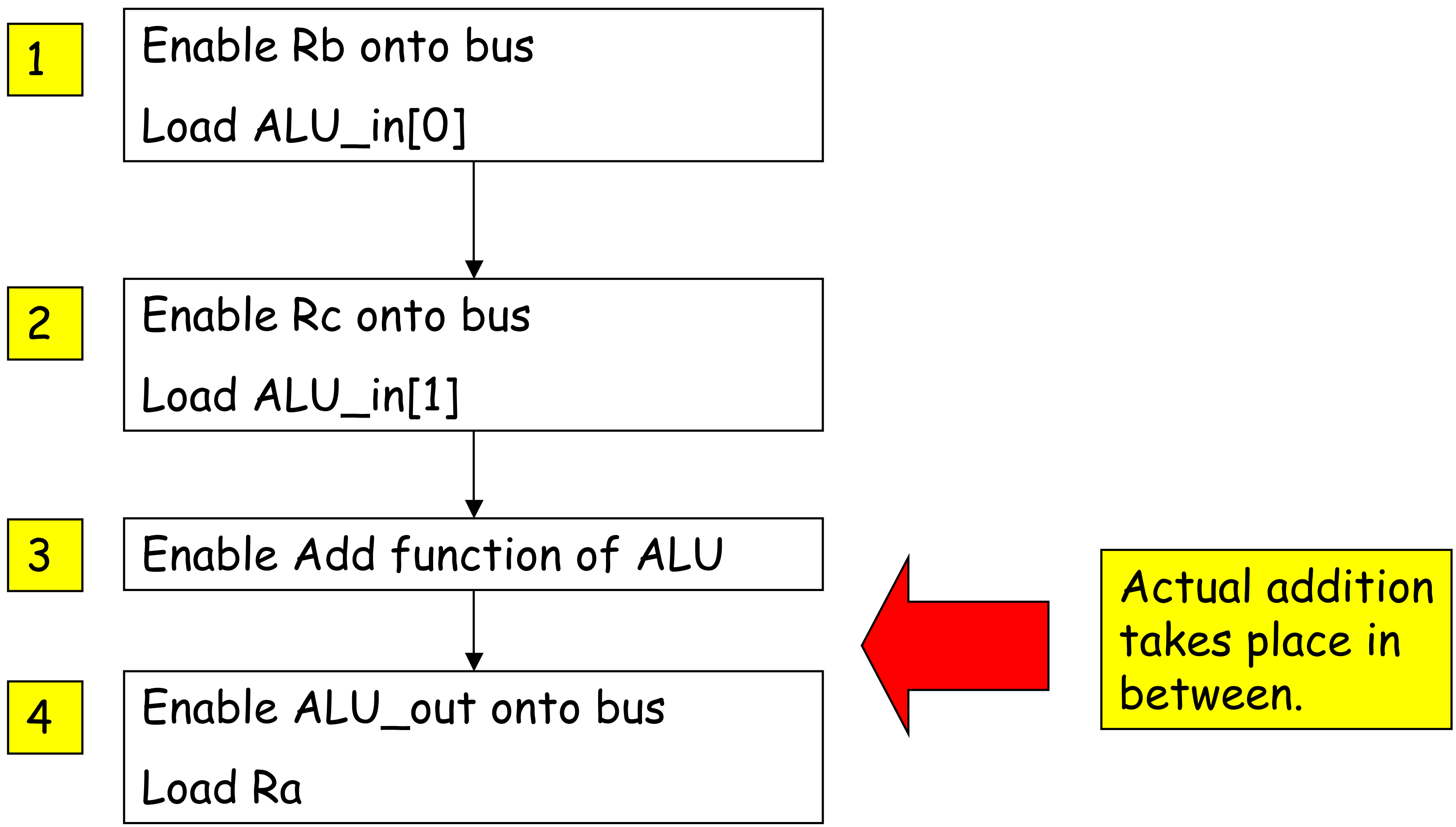


3'

Increment IP



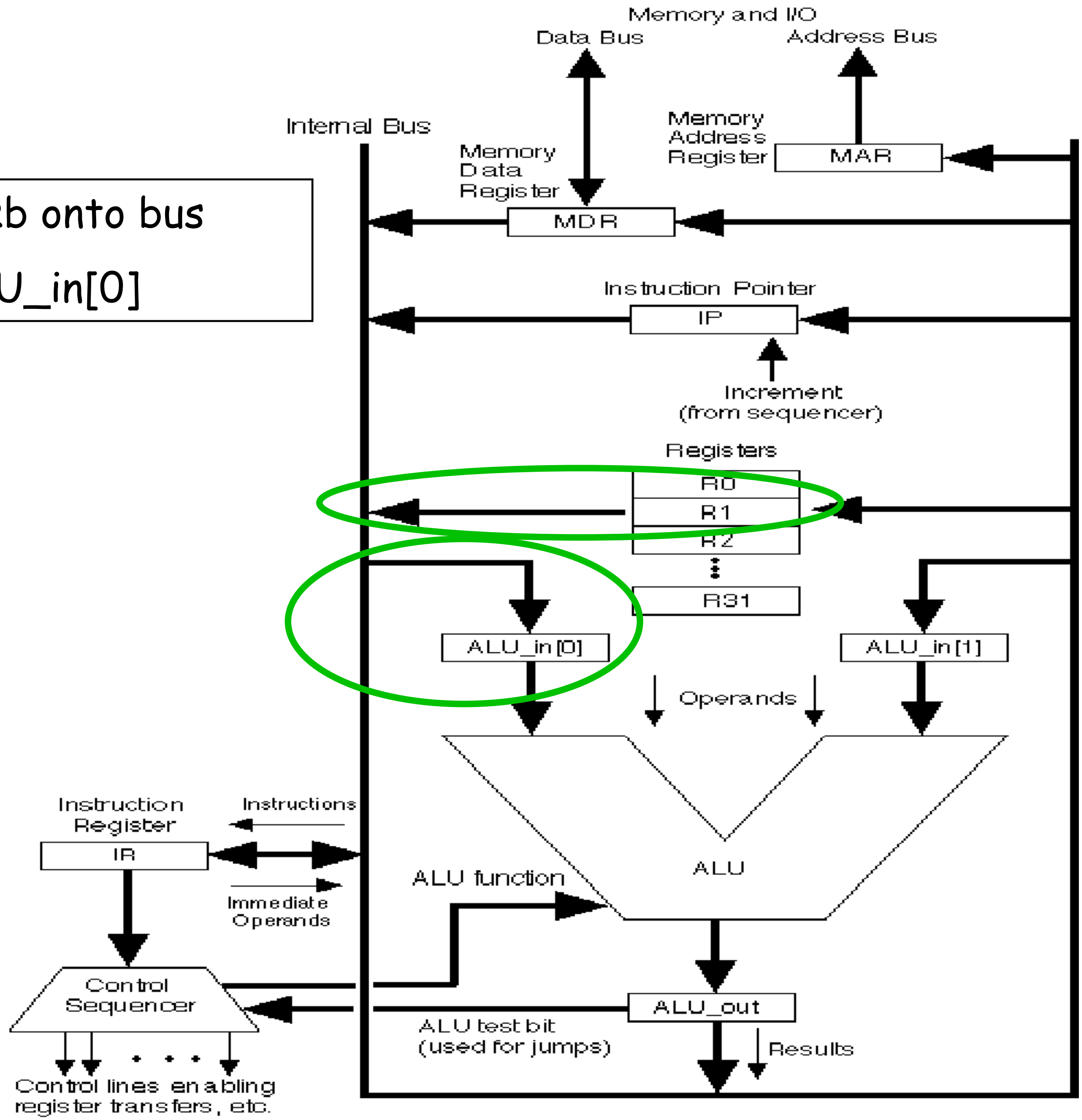
Control Subsequence Example: Add Ra Rb Rc



1

Enable Rb onto bus

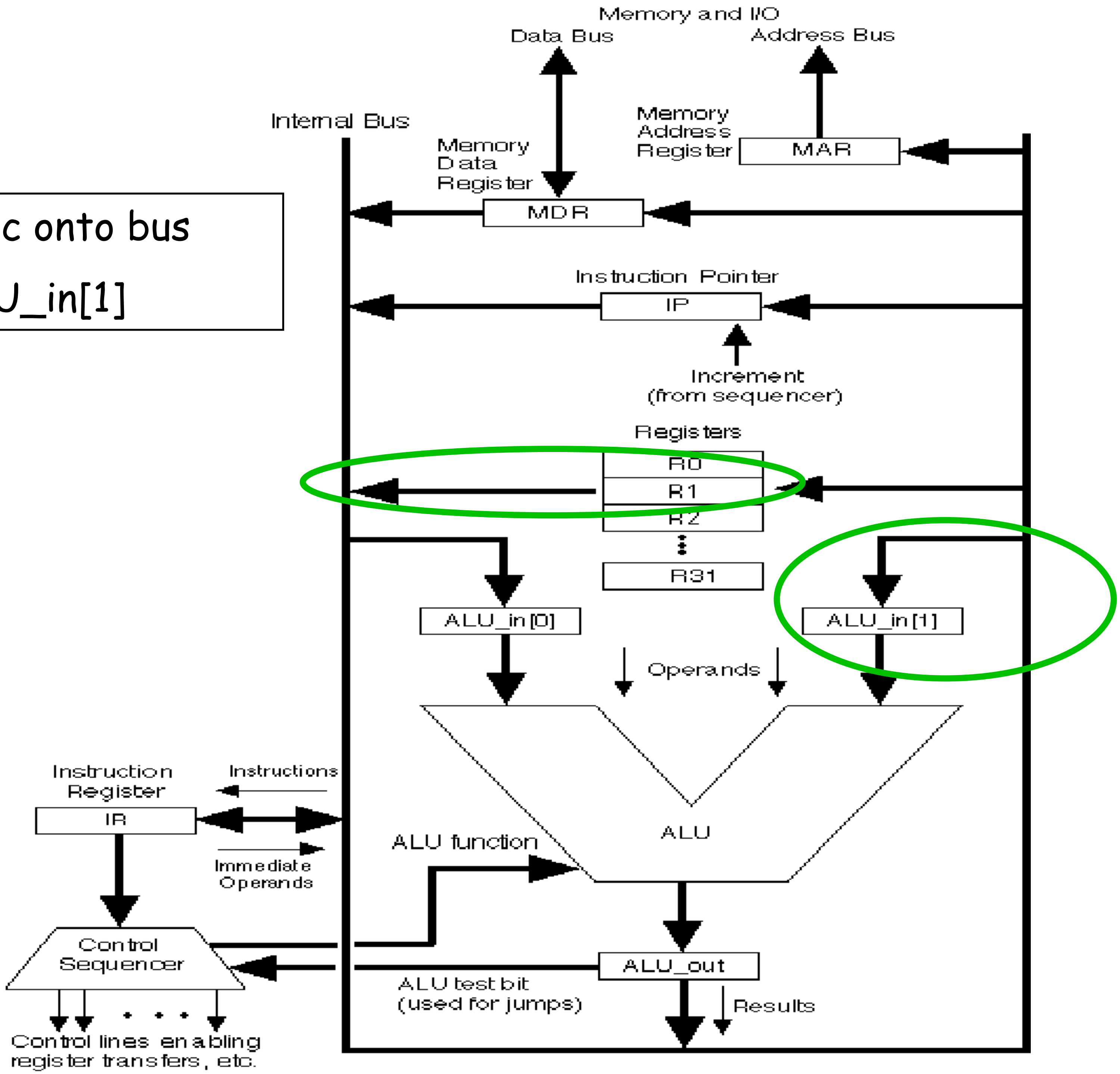
Load ALU_in[0]



2

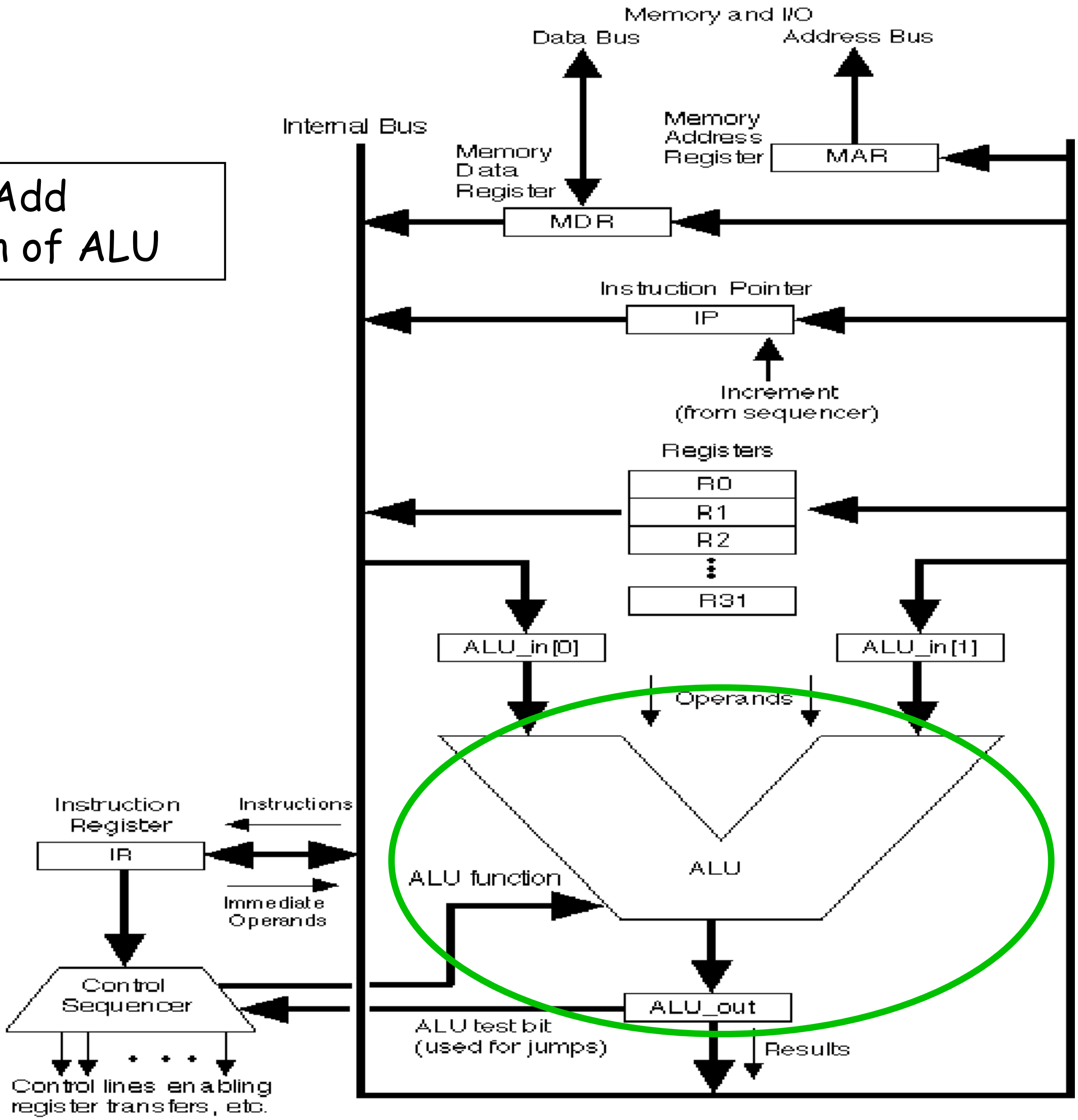
Enable Rc onto bus

Load ALU_in[1]



3

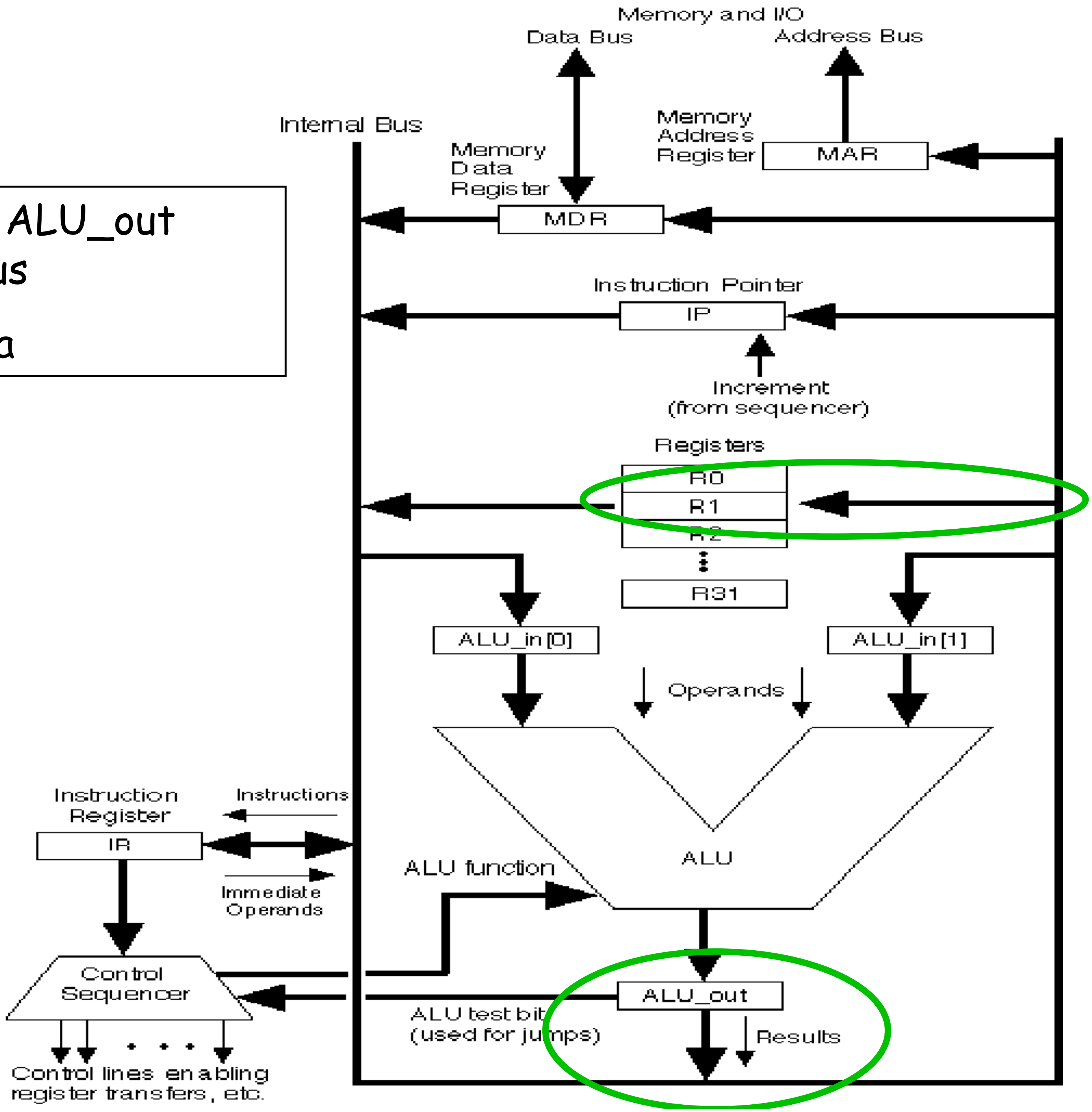
Enable Add function of ALU



4

Enable ALU_out
onto bus

Load Ra



Exercise

- What would the instruction subsequences be for:
  - aim (add immediate)
  - load
  - store
  - jeq (jump-if-equal)

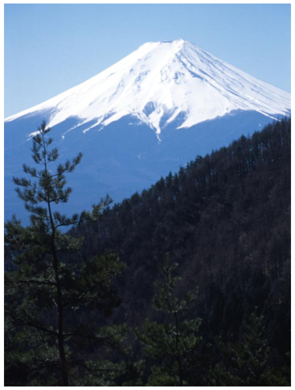
Machine-Level Programming

- Programming of the bare machine is typically done in "assembly language"
- One line of assembly language is roughly equal to one machine instruction
- A program, the "assembler", allows use of symbolic identifiers for addresses.

Programming in Assembly Language

- Programming in assembly language reminds me of the Japanese saying about climbing Mt. Fuji:

  "To never have climbed Mt. Fuji is to be a fool. Only a (bigger) fool would climb Mt. Fuji more than once."



ISCAL
ISC Assembly Language

- See http://www.cs.hmc.edu/~keller/isc/
- Free-form input, but generally format line-by-line
- Regular instructions
  - *Ra*, *Rb*, *Rc* are register names
  - *C* is a constant
    - **lim** *Ra C*
    - **add** *Ra Rb Rc*
    - **copy** *Ra Rb*
    - **shl** *Ra Rb*
    - **load** *Ra Rb*
    - etc.

ISCAL

- Assembler directives: not instructions to computer, but rather tell assembler what to do:
  - **define** *Identifier Value* — Defines *Identifier* to have *Value.*
  - **use** *Identifier* — Defines *Identifier* to name an unused register.
  - **origin** *Value* — Begin loading instructions at specified memory location. The location counter is incremented as loading progresses.
  - **label** *Identifier* — Associates current instruction location with *Identifier.*

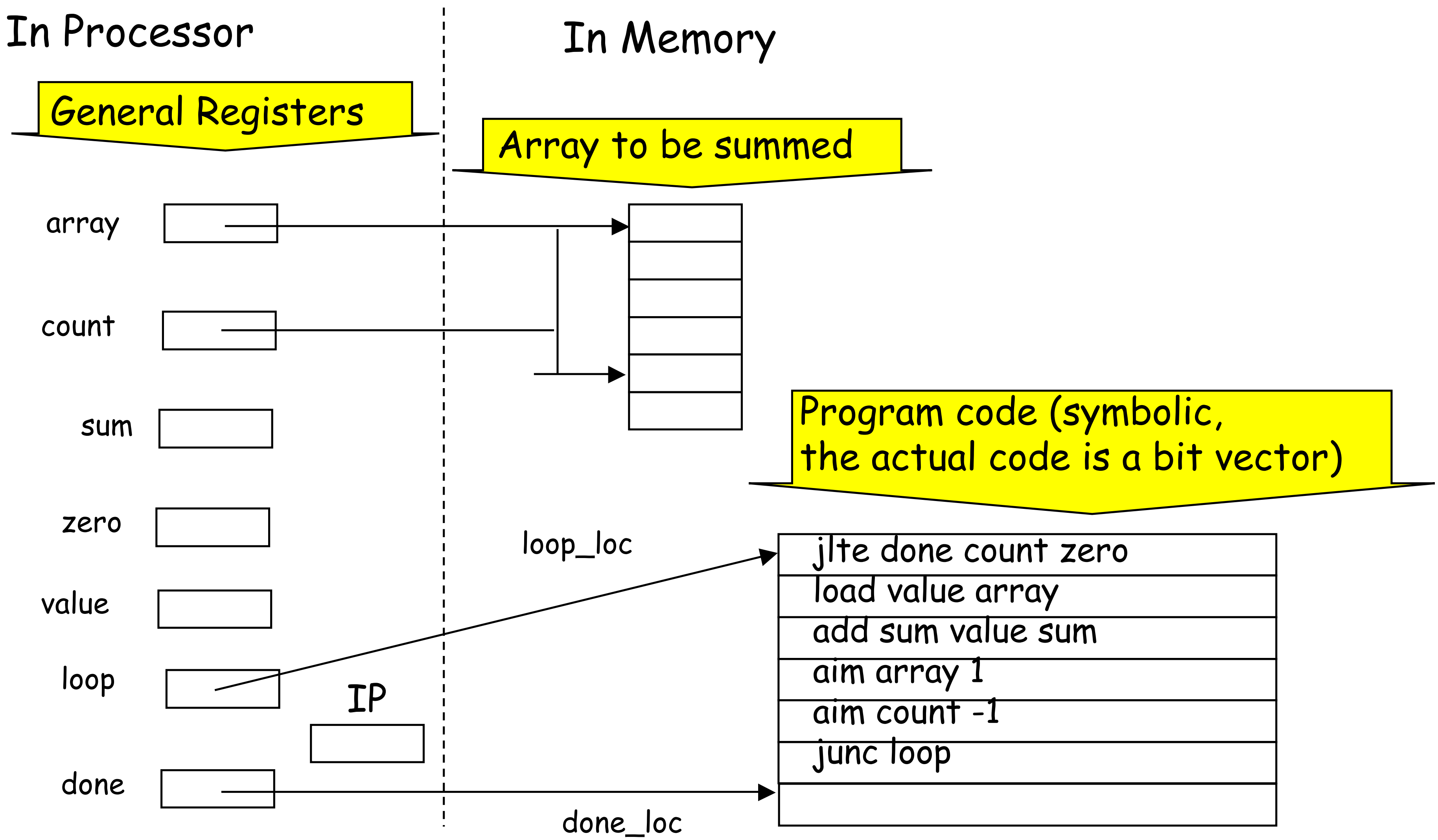
ISCAL code for summing an array

```
   use array  use count             // array base and count
   use sum    use zero   use value  // local registers
   use loop   use done              // address registers
```

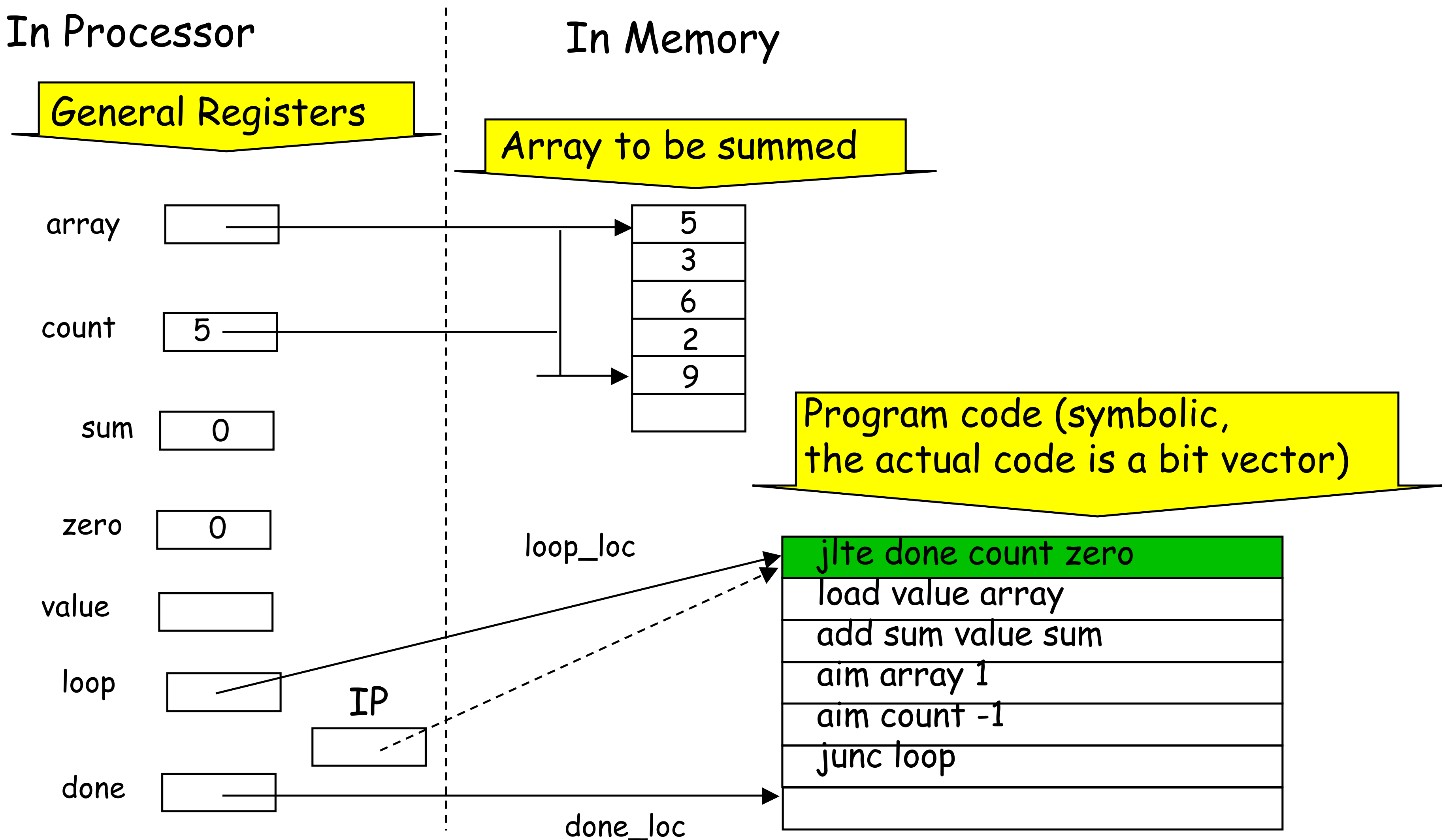
… insert code to load array and count values …

```
    lim sum  0          // initialize sum
    lim zero  0         // comparison value
    lim done done_loc   // address of instruction following
    lim loop loop_loc   // address of next instruction

label loop_loc

    jlte done count zero // jump if <= 0
    load value array     // load register next array value
    add sum value sum    // add the next number to the sum
    aim array 1          // add 1 to the array address
    aim count -1         // add -1 to the count
    junc loop            // go back and compare

label done_loc
```

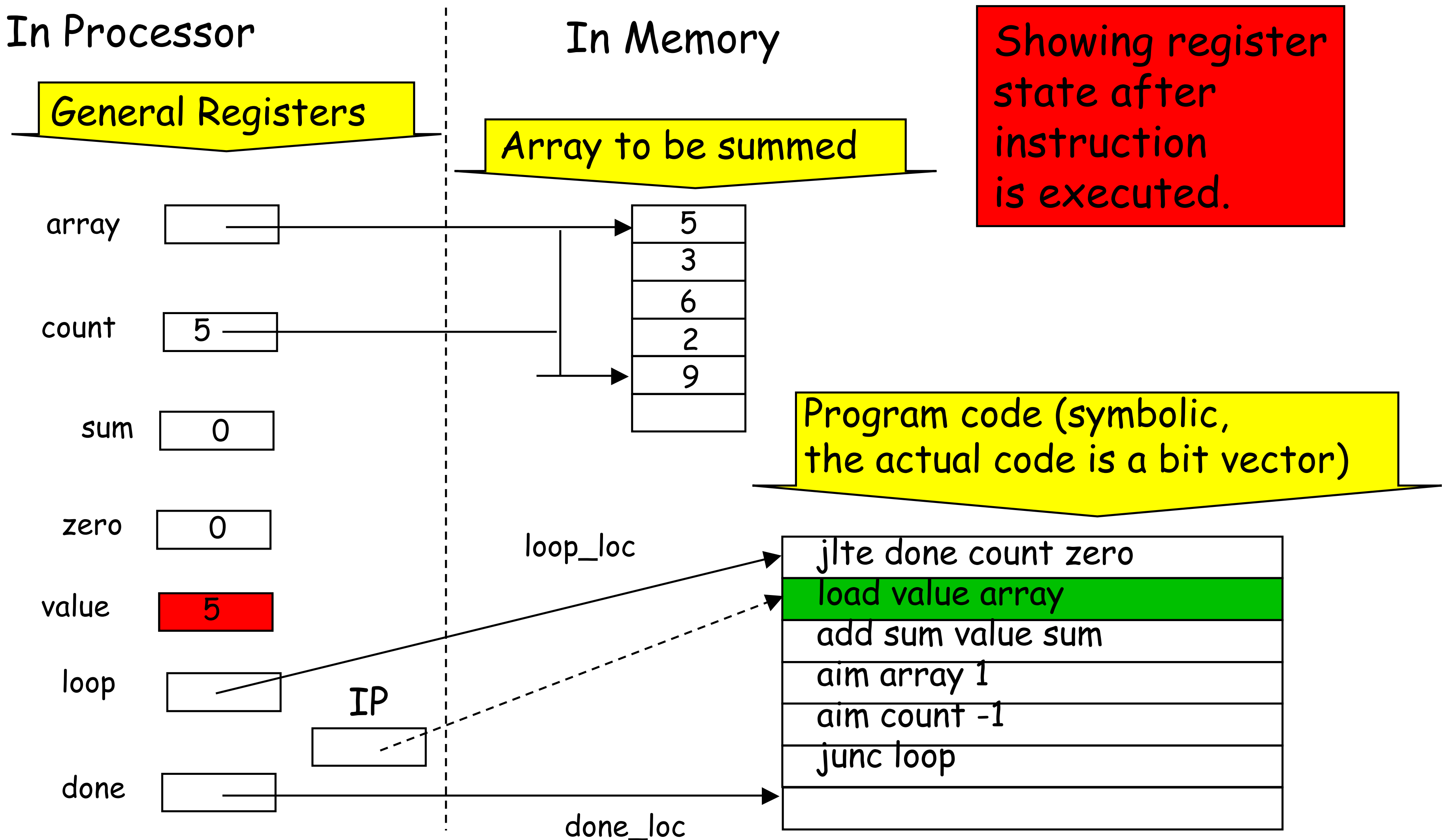
Array Summation



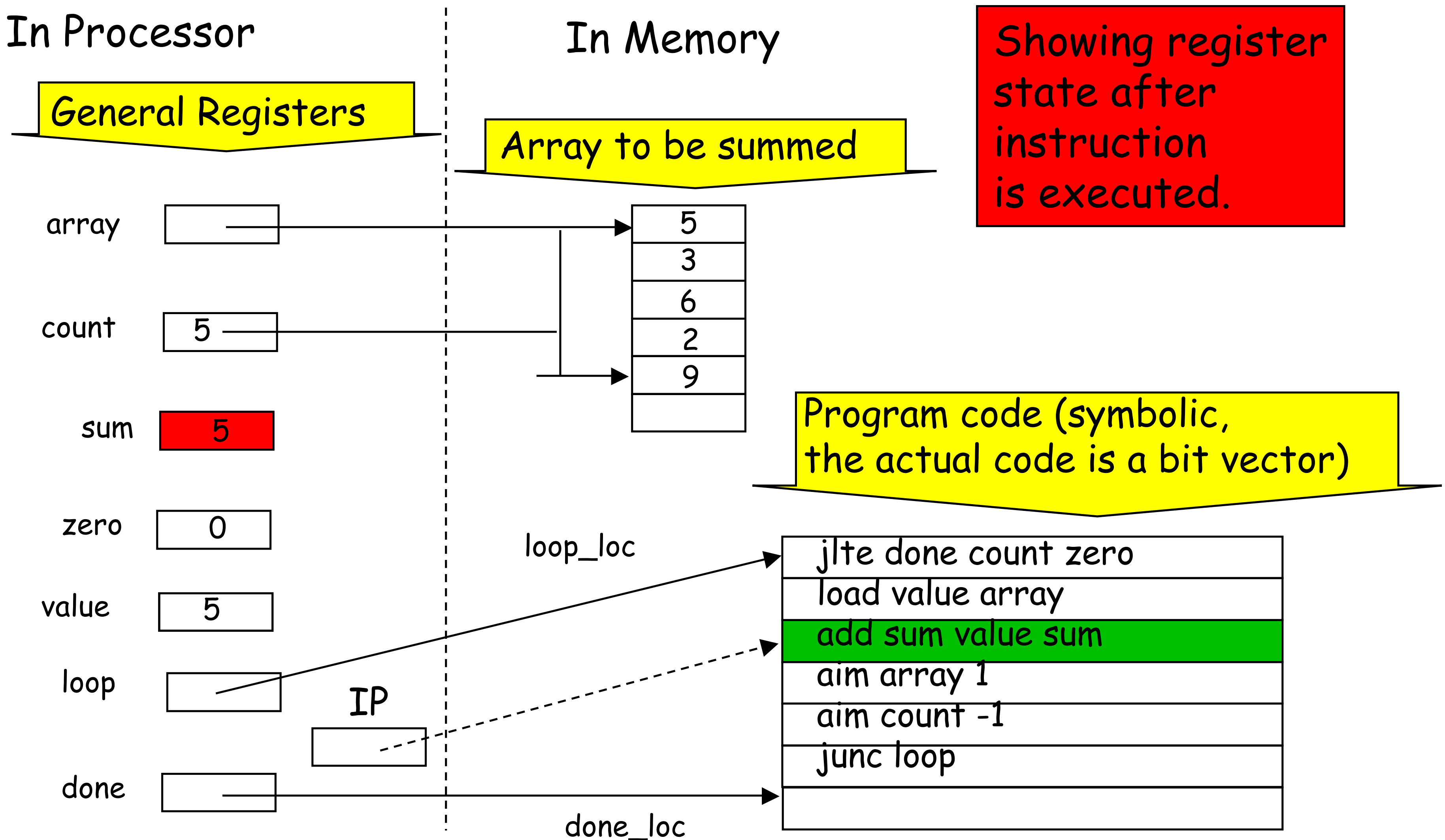
Array Summation



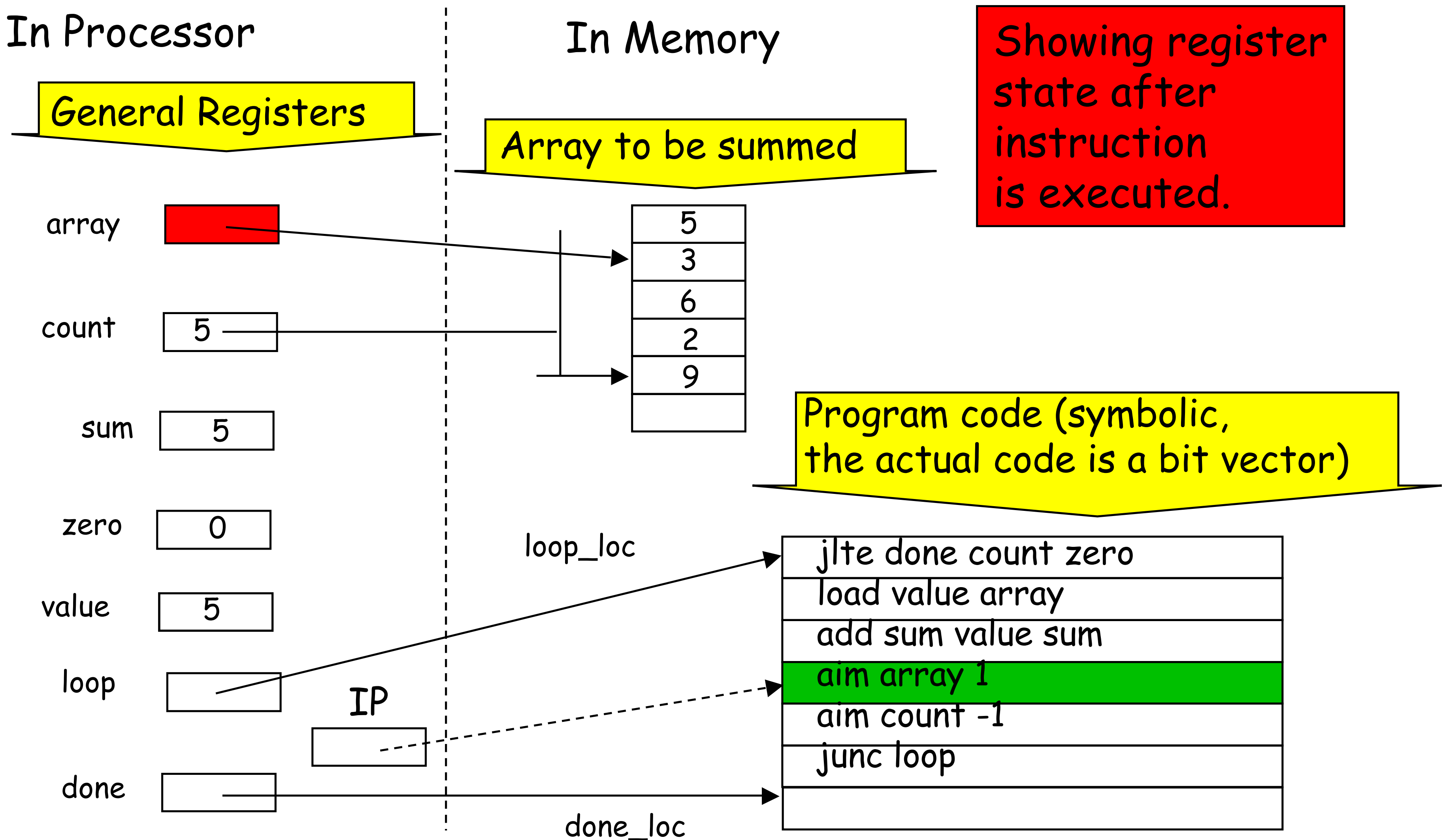
Array Summation



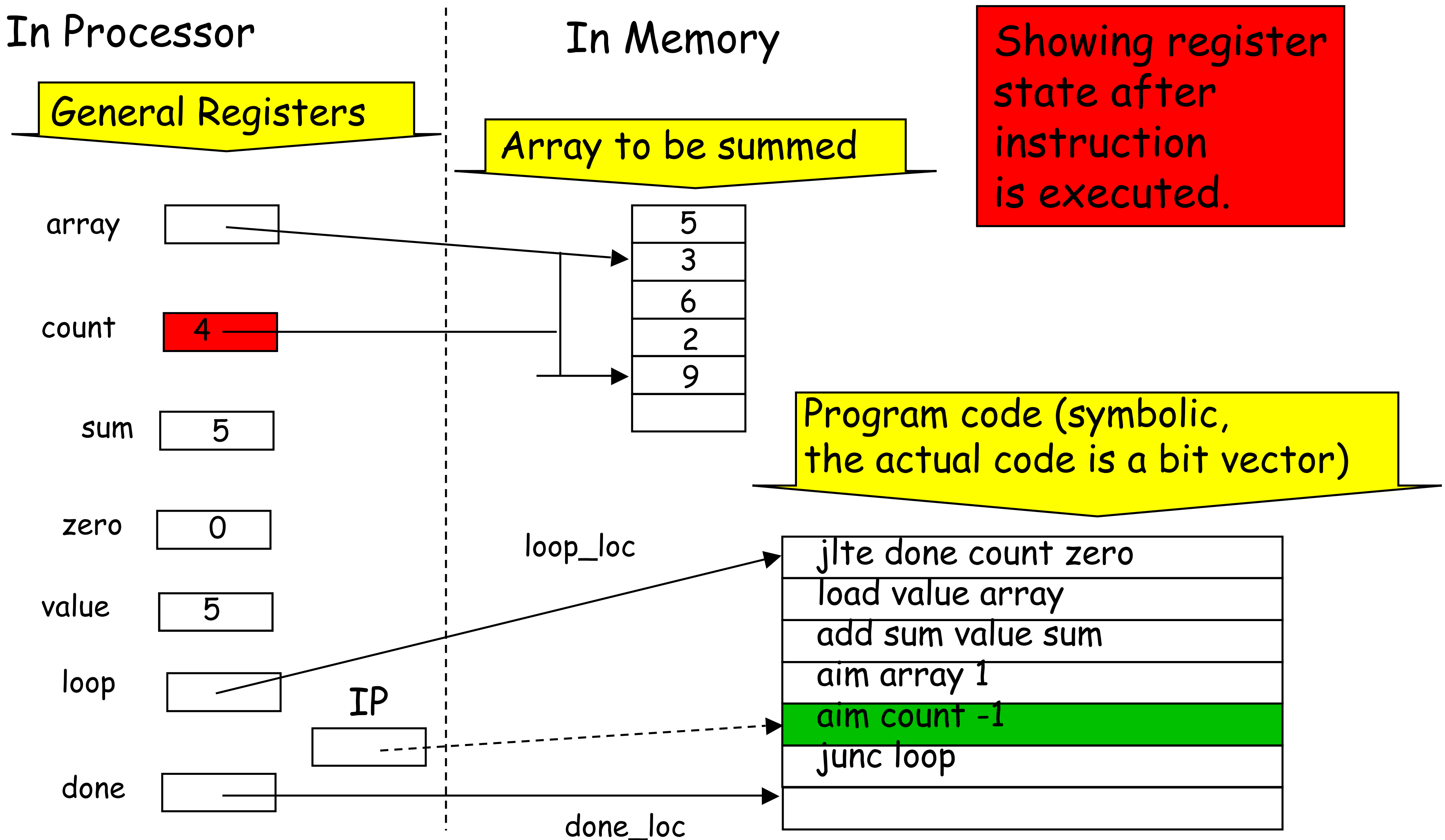
Array Summation



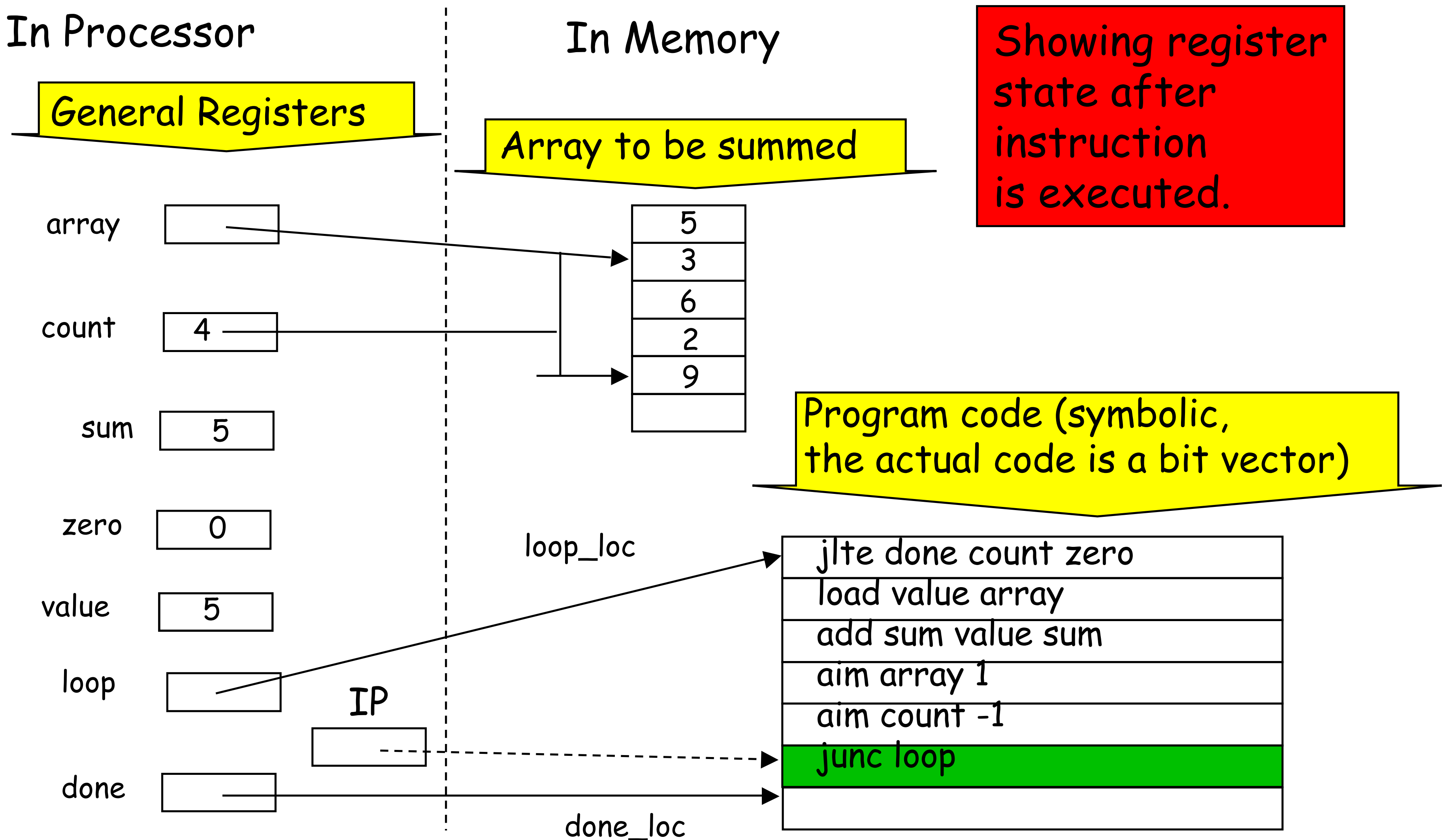
Array Summation



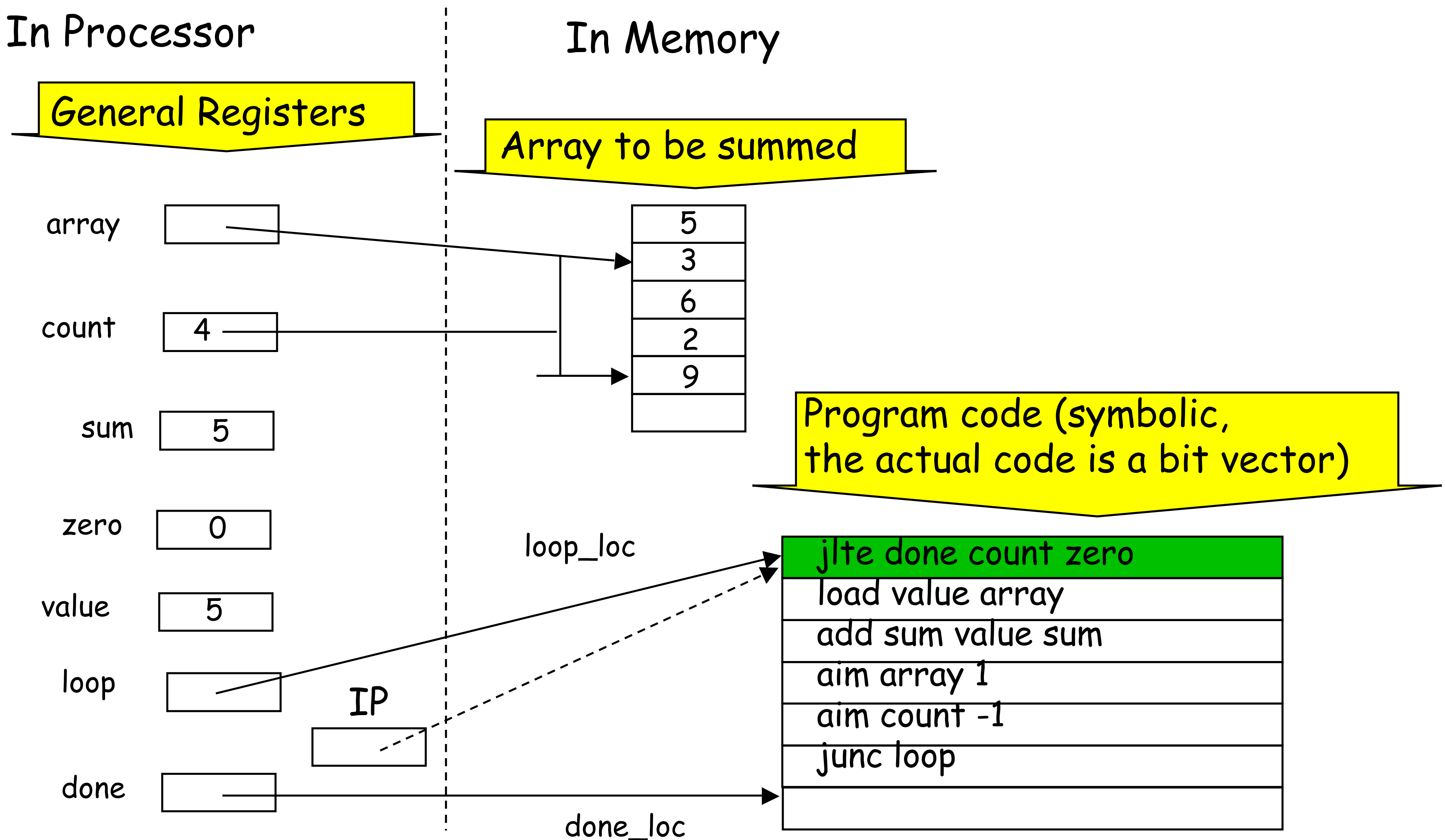
Array Summation



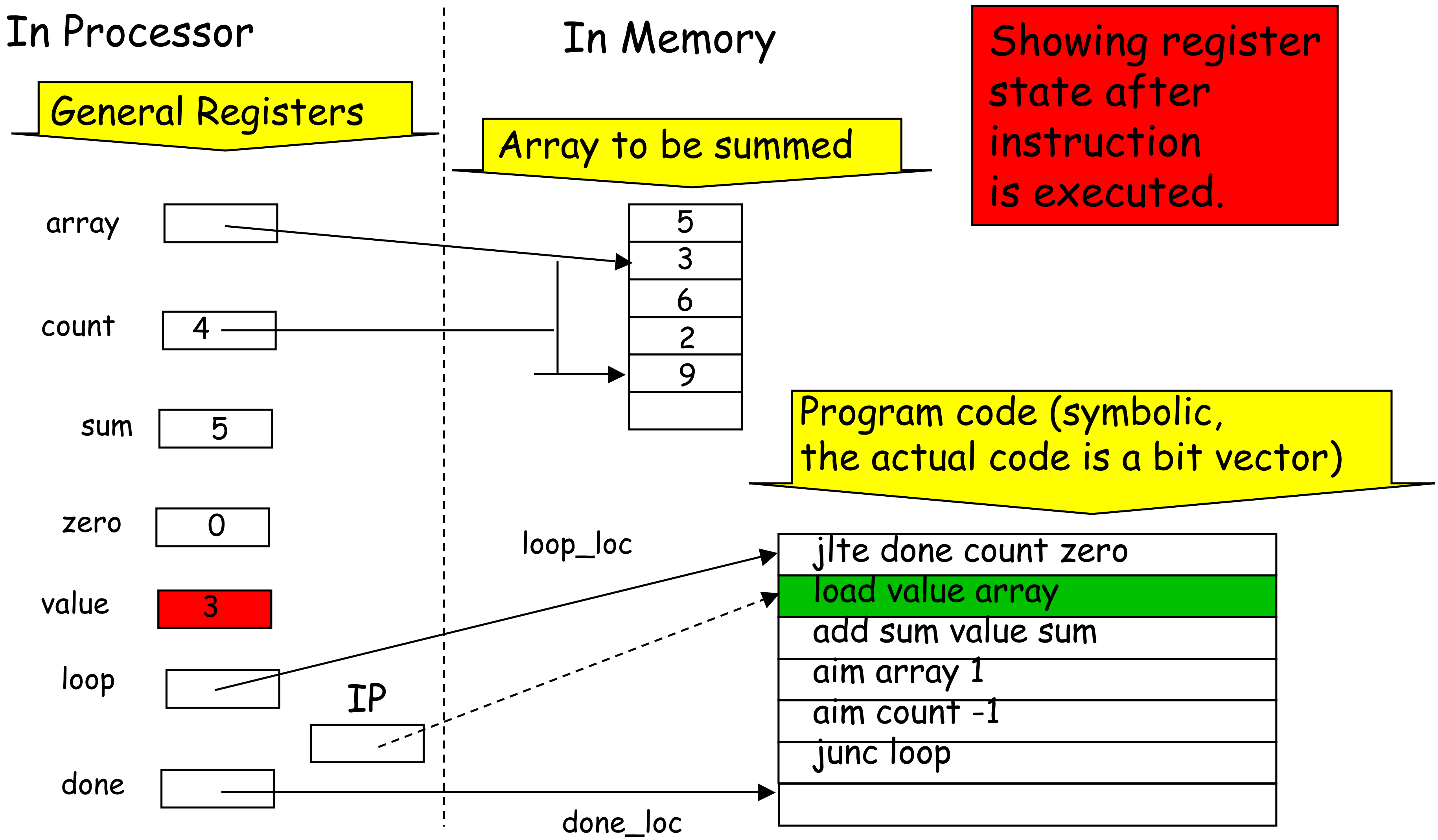
Array Summation



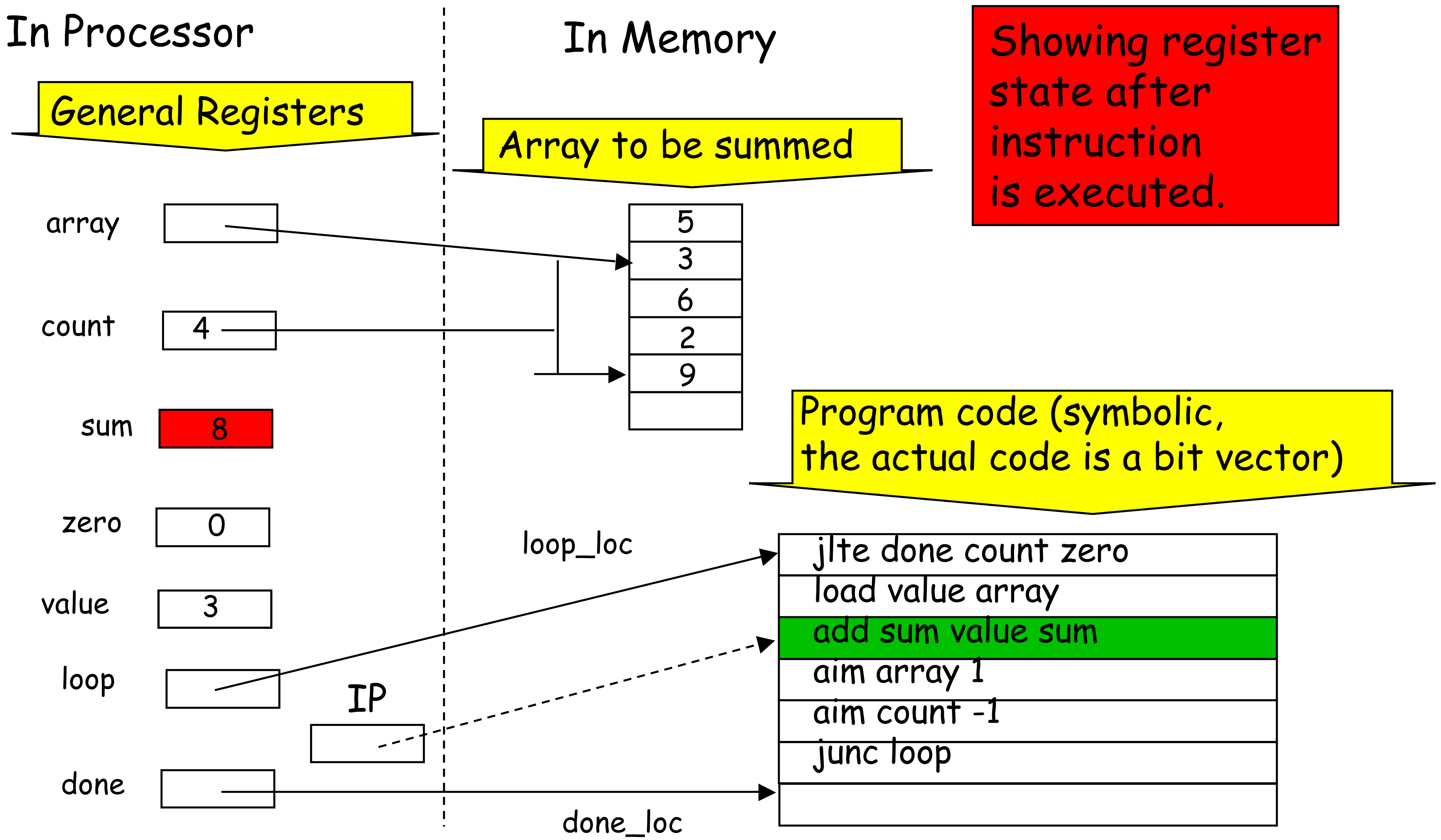
Array Summation



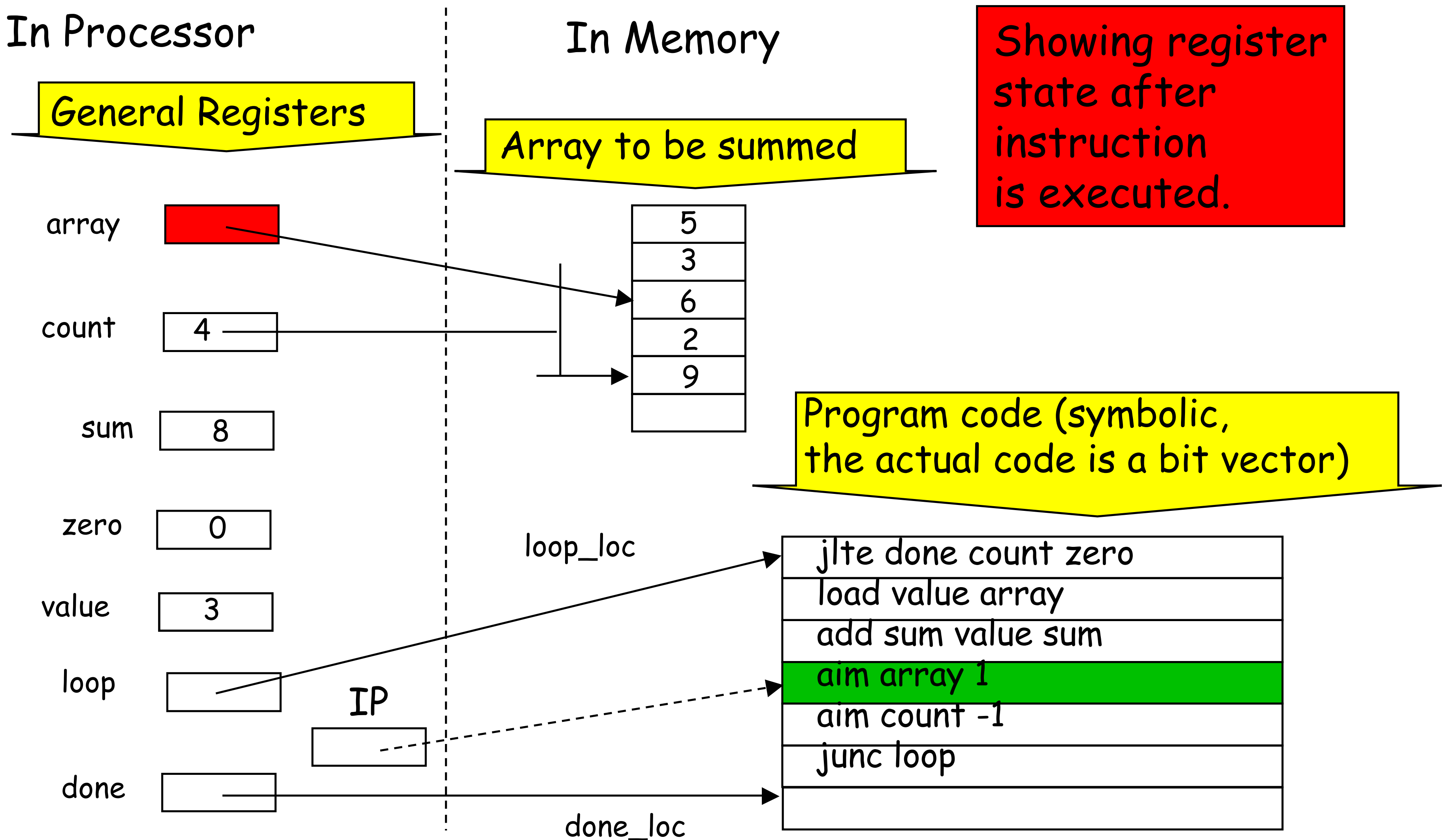
Array Summation



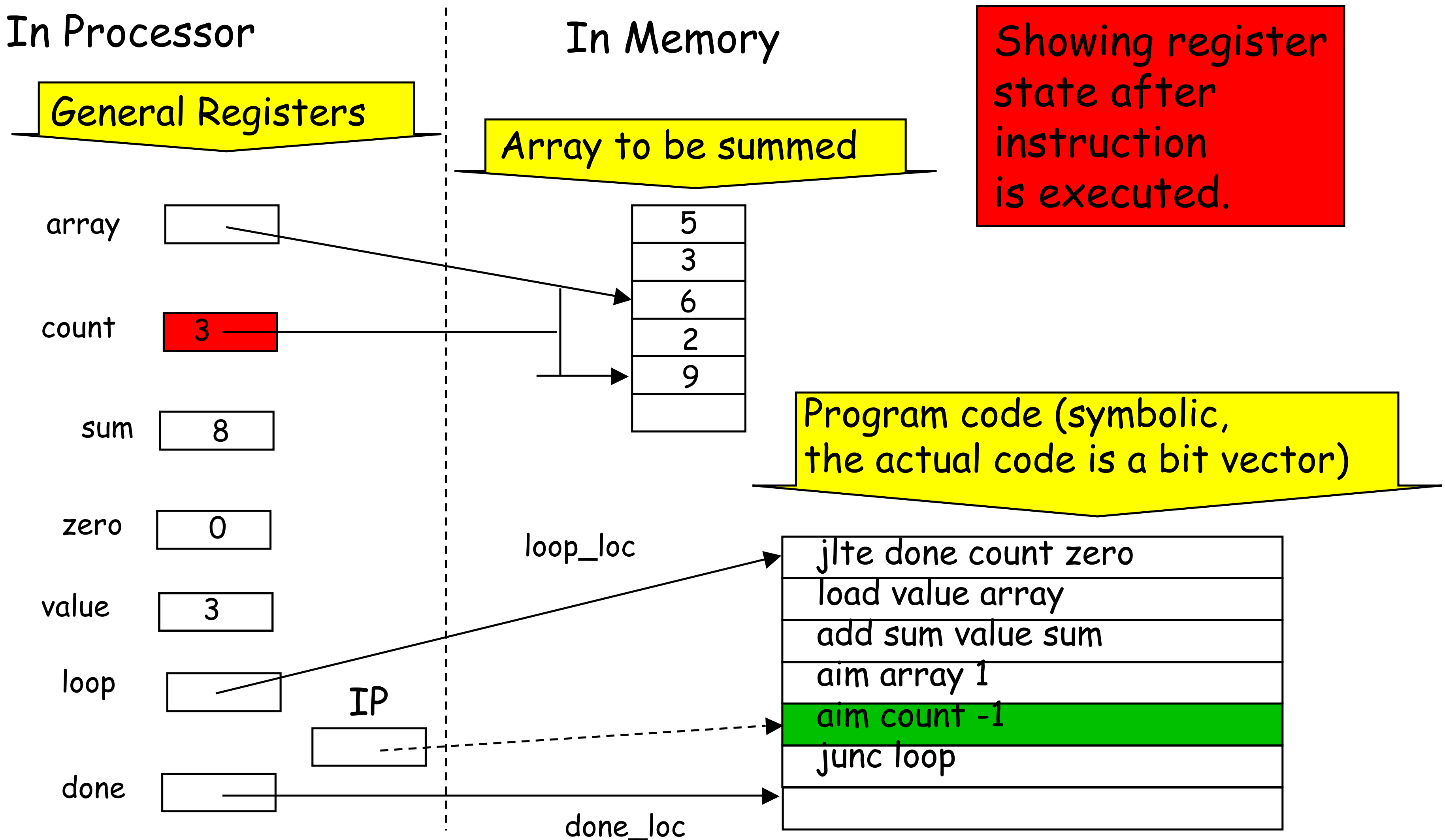
Array Summation



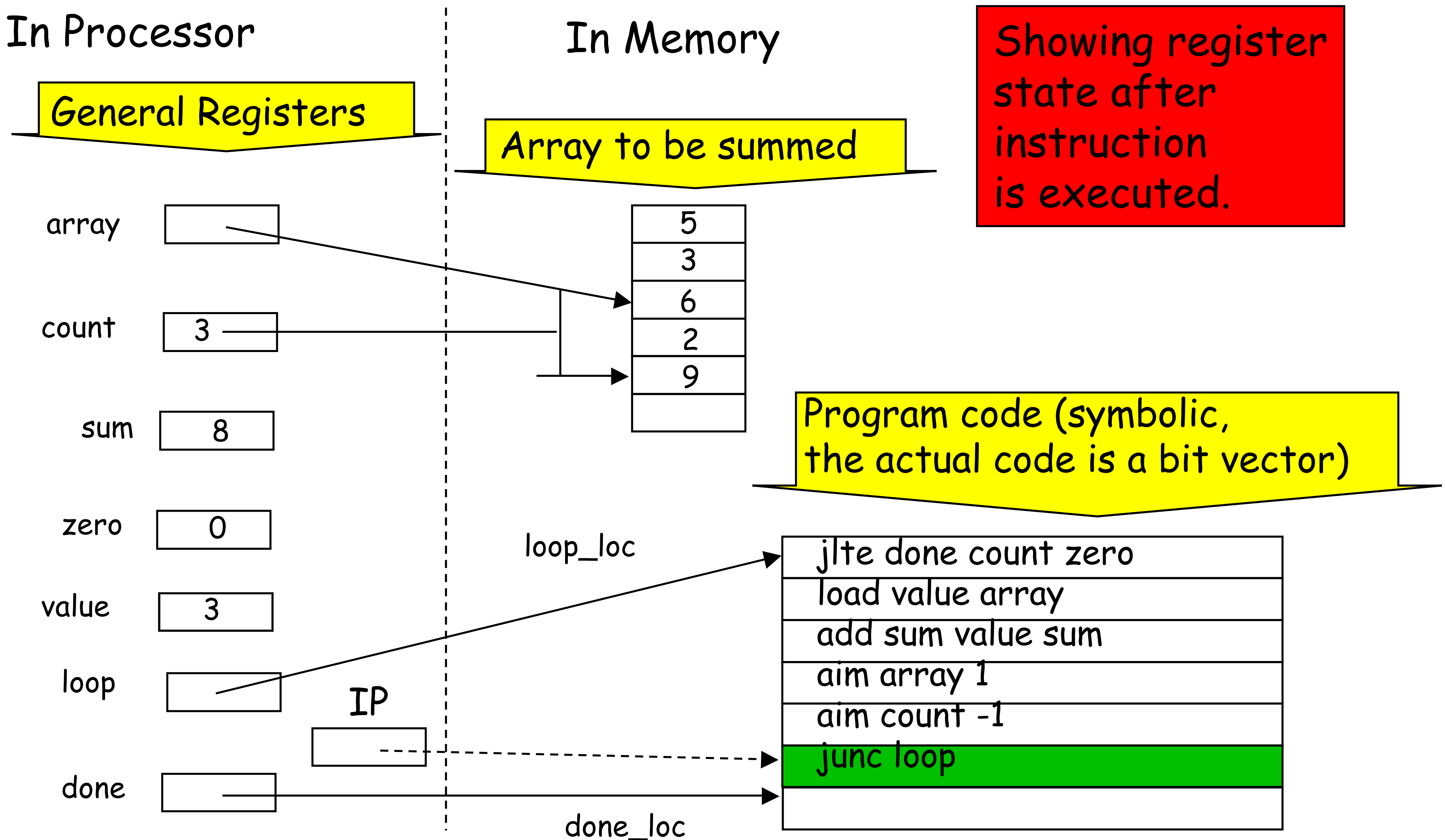
Array Summation



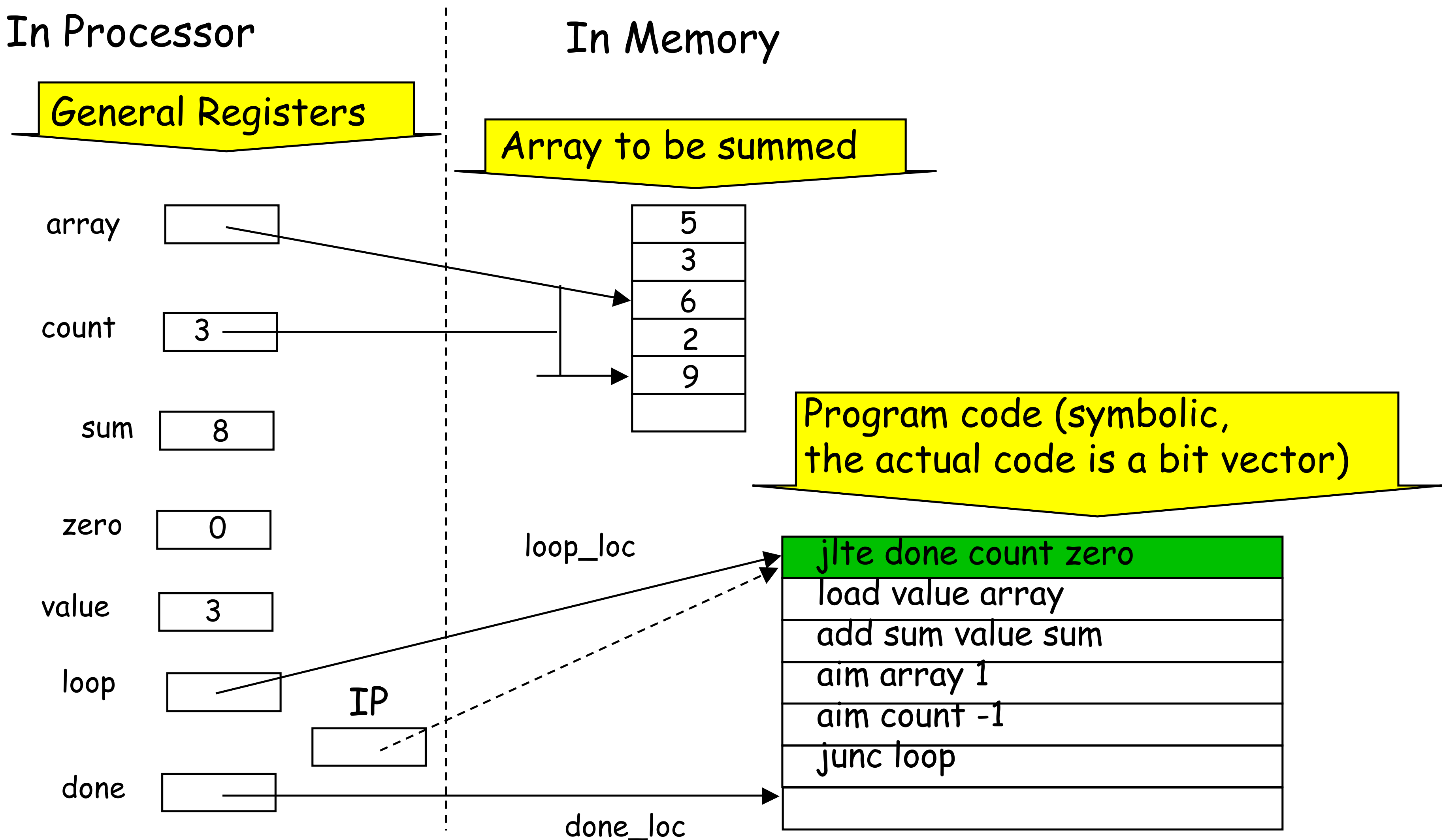
Array Summation



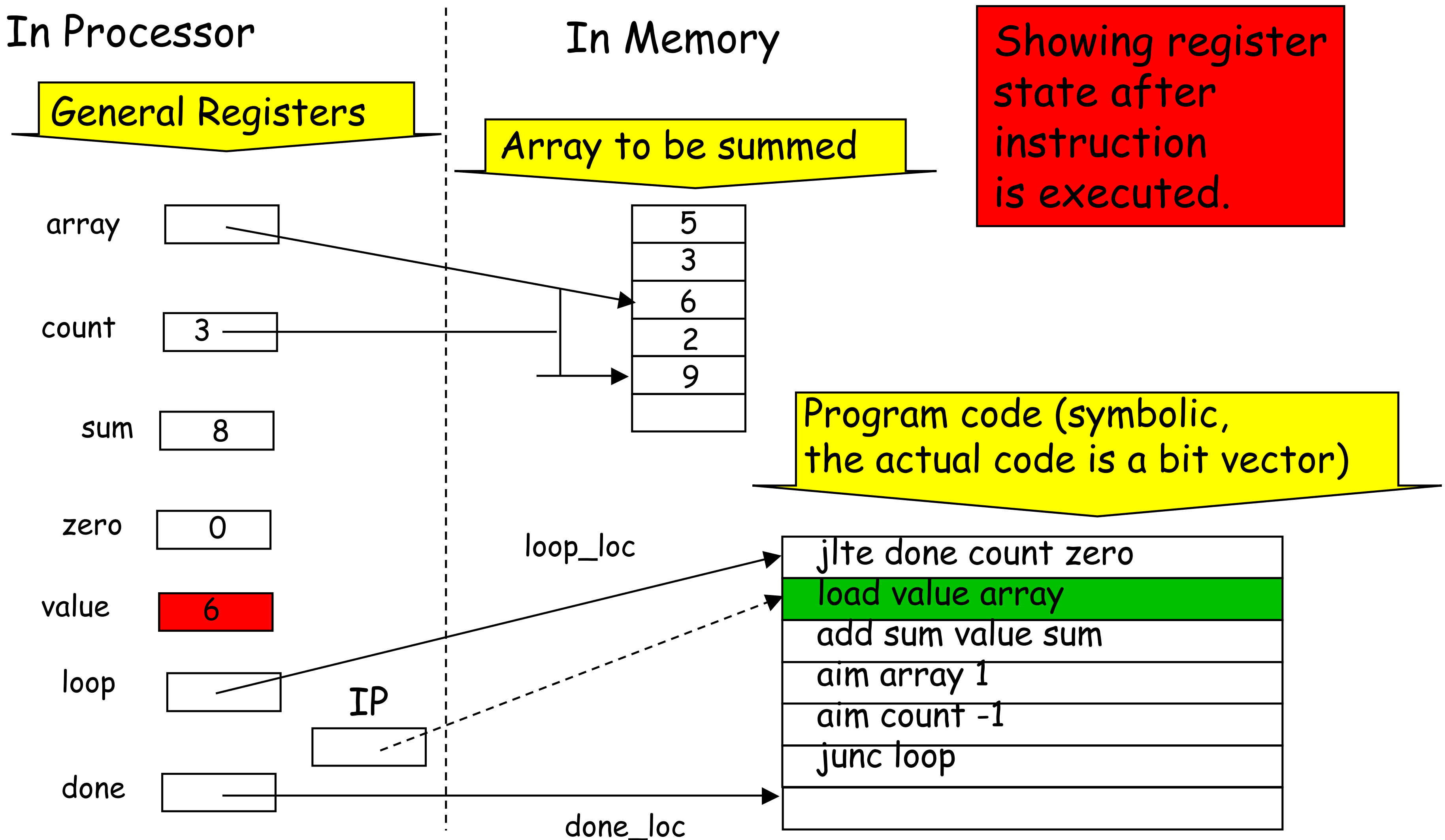
Array Summation



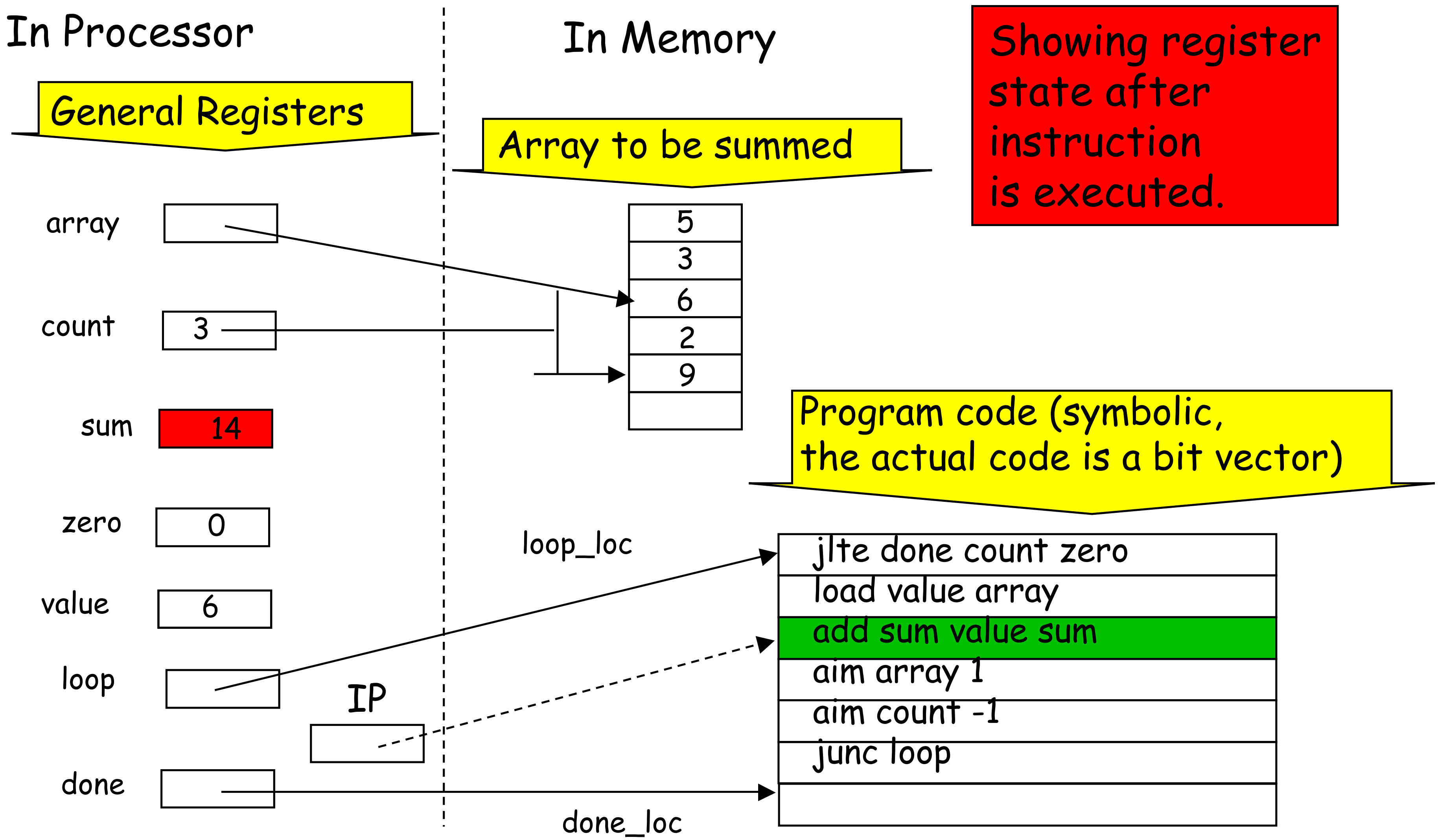
Array Summation



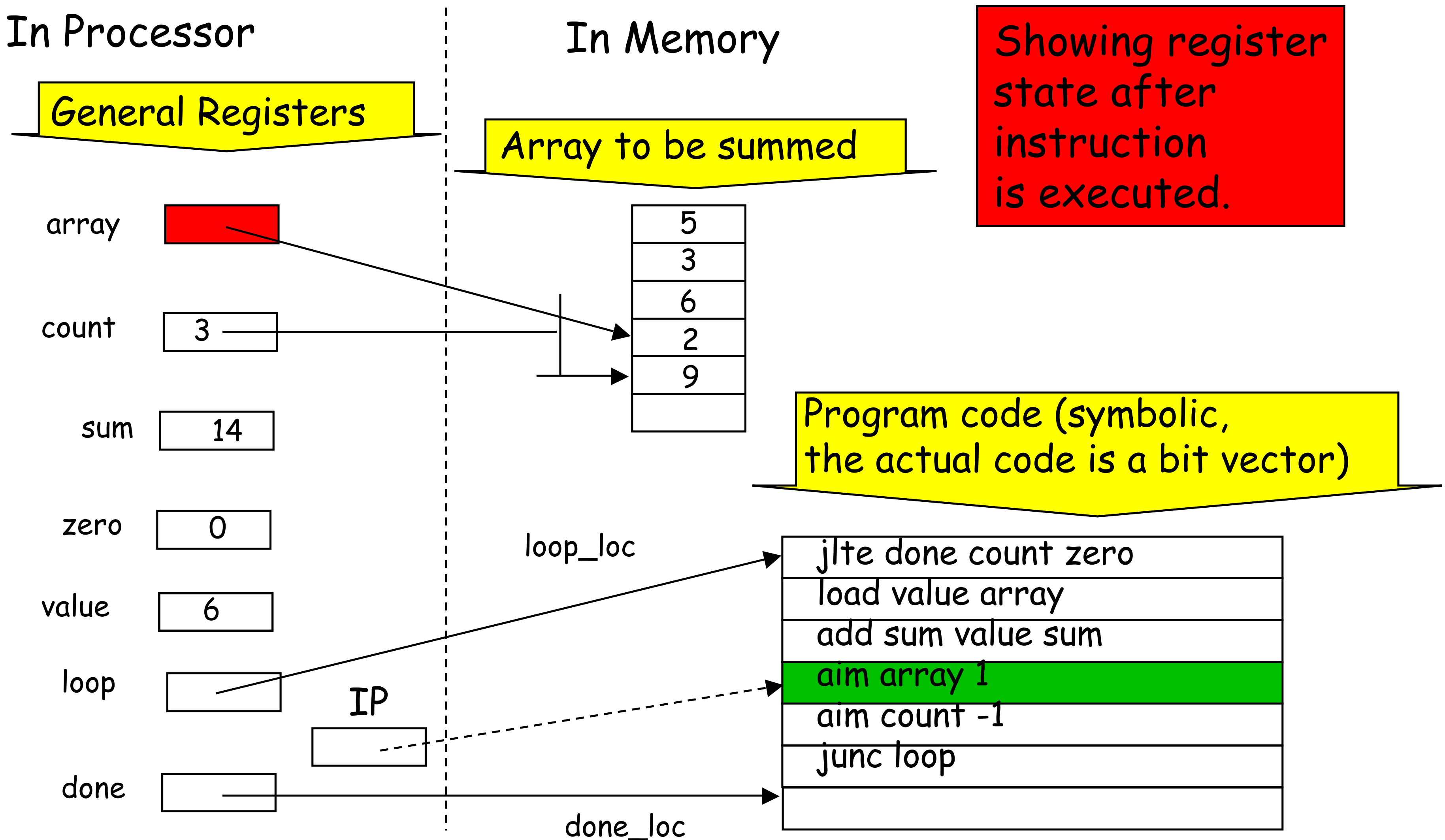
Array Summation



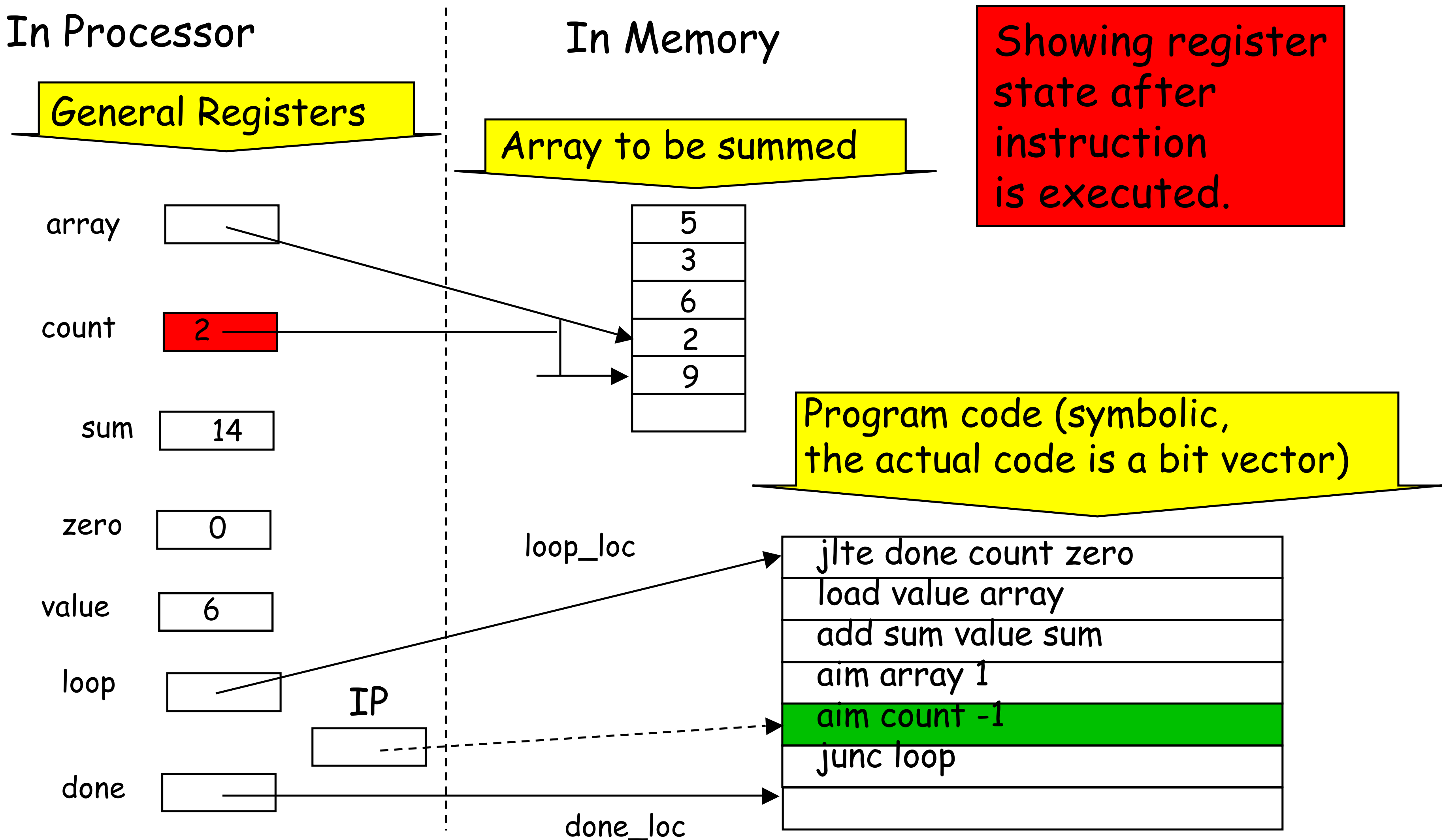
Array Summation



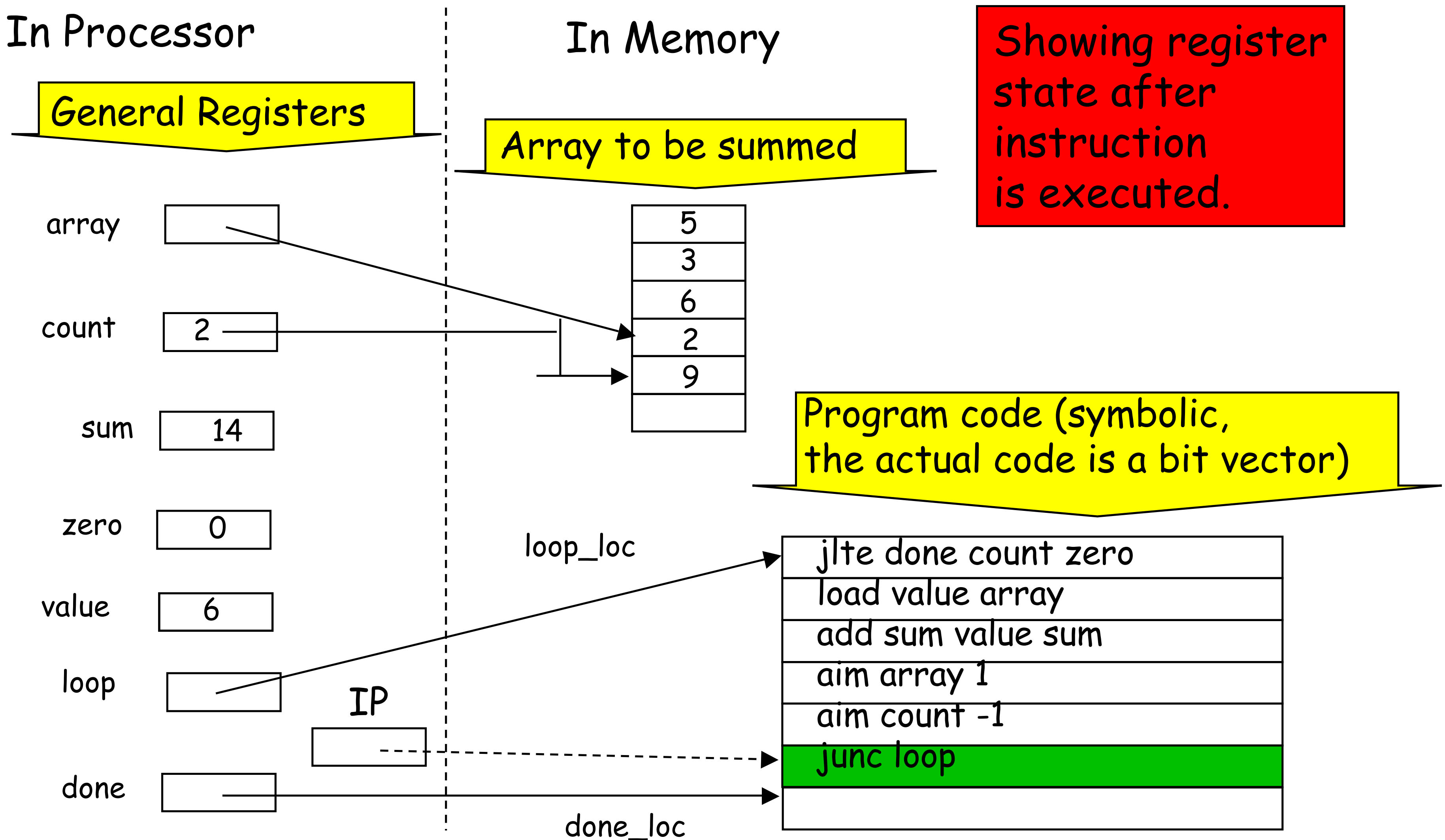
Array Summation



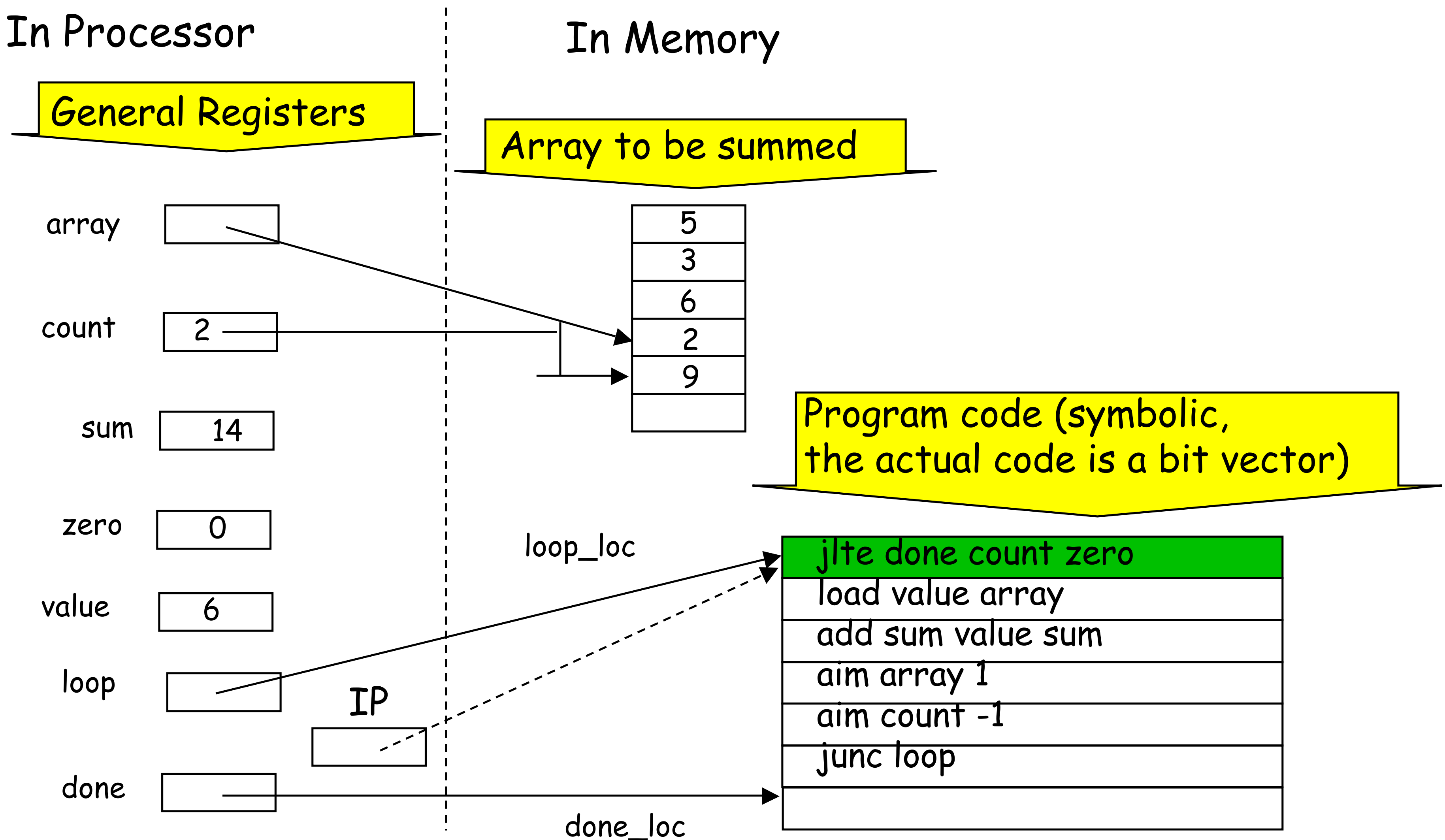
Array Summation



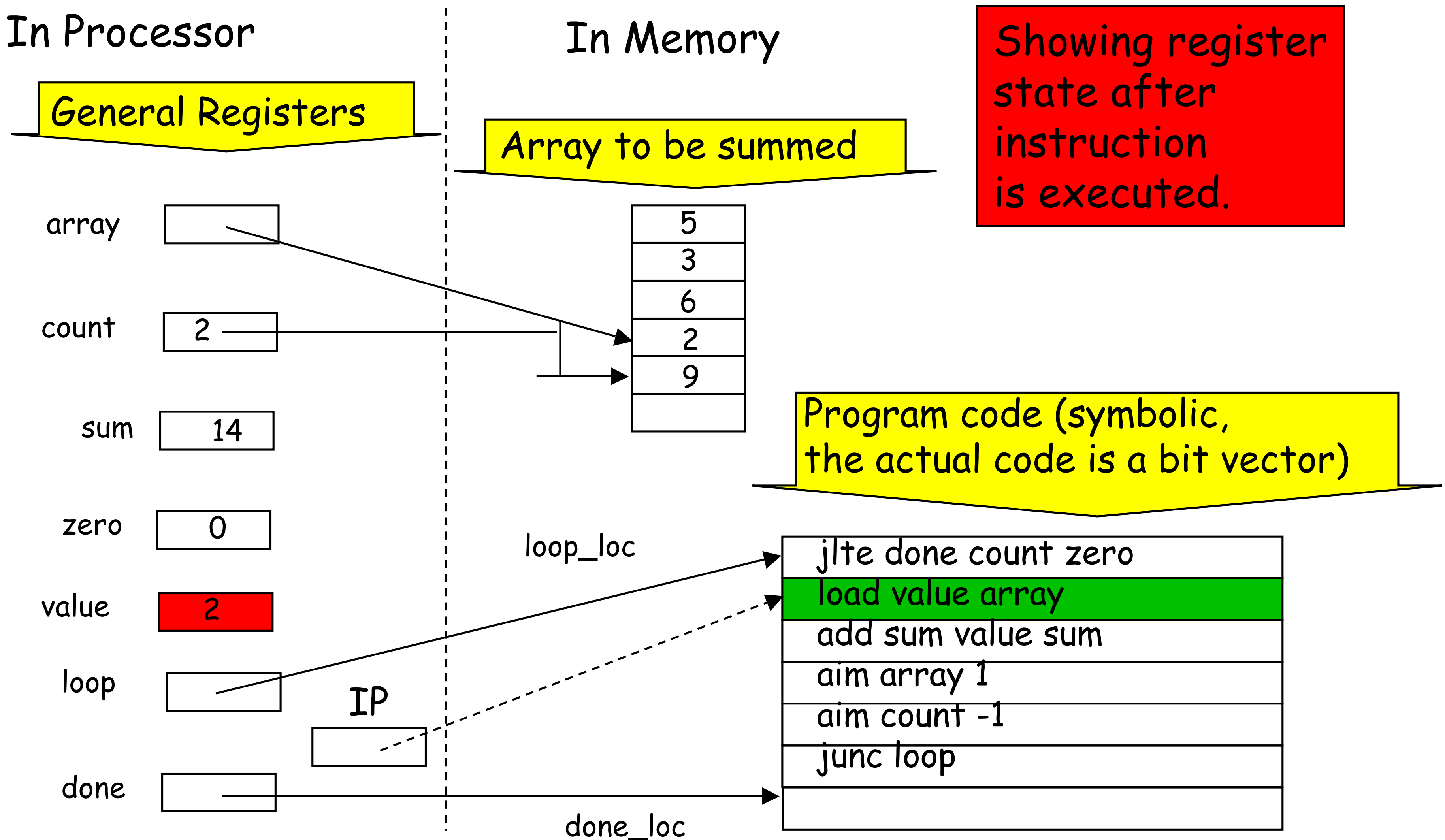
Array Summation



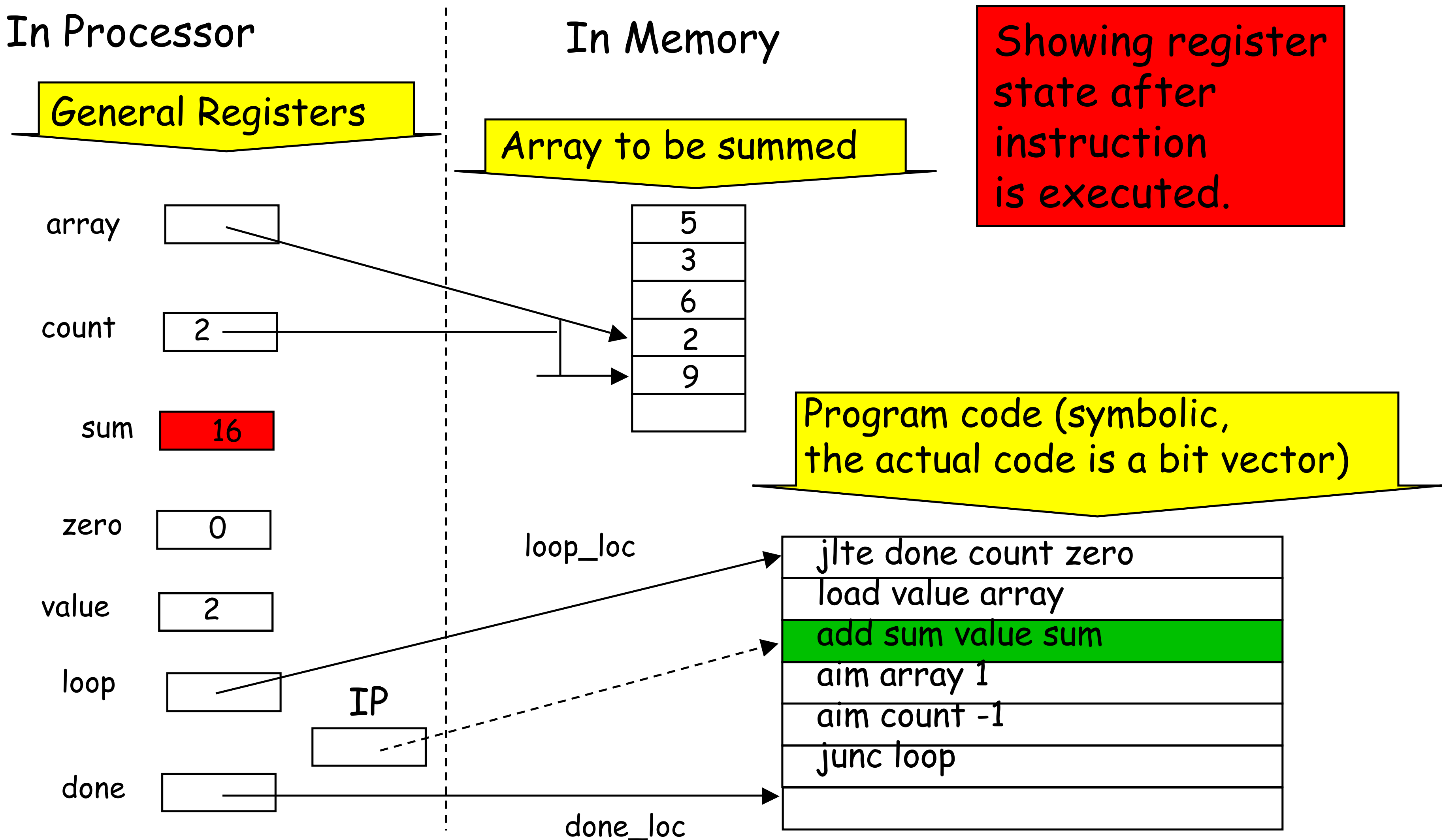
Array Summation



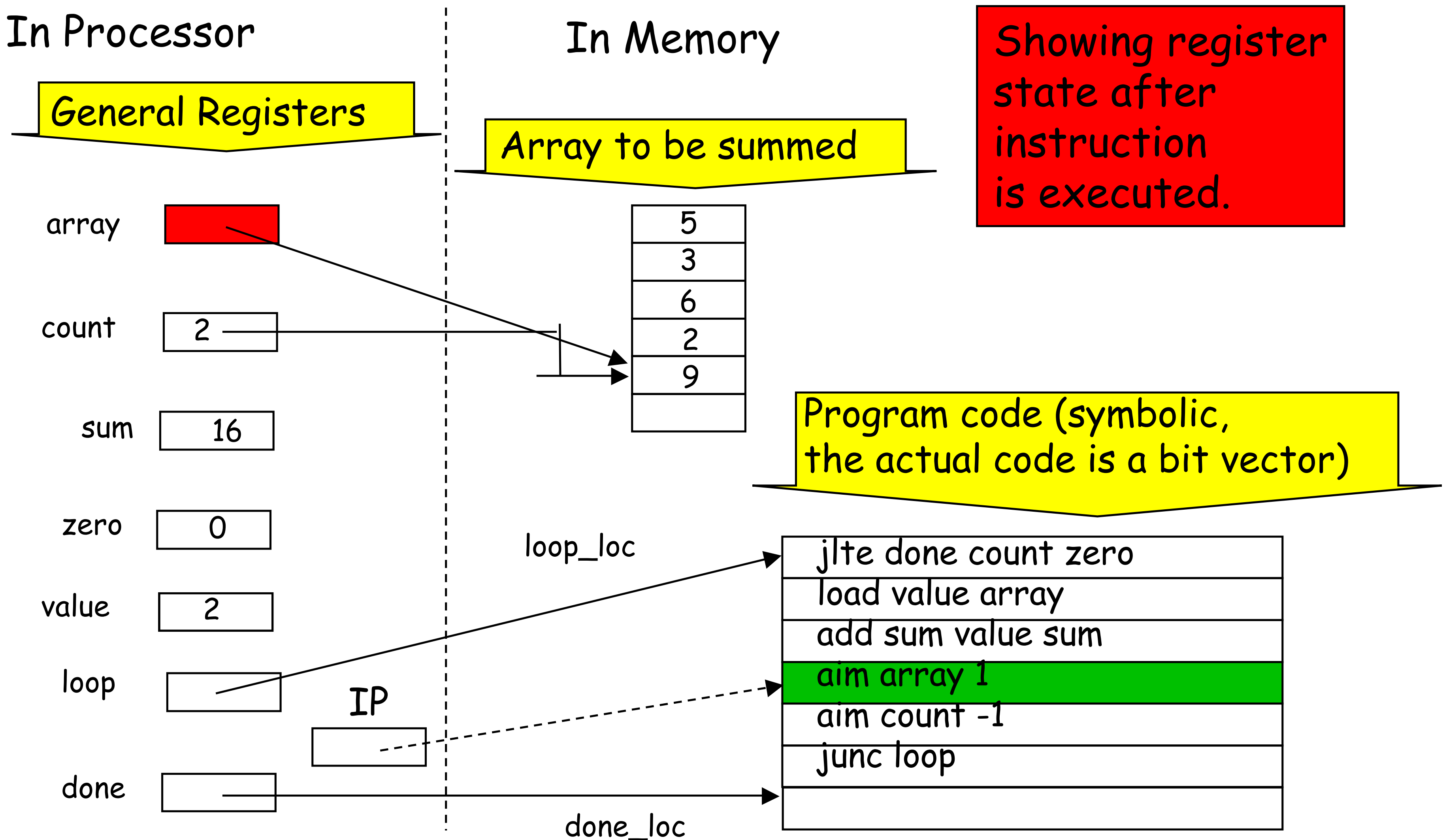
Array Summation



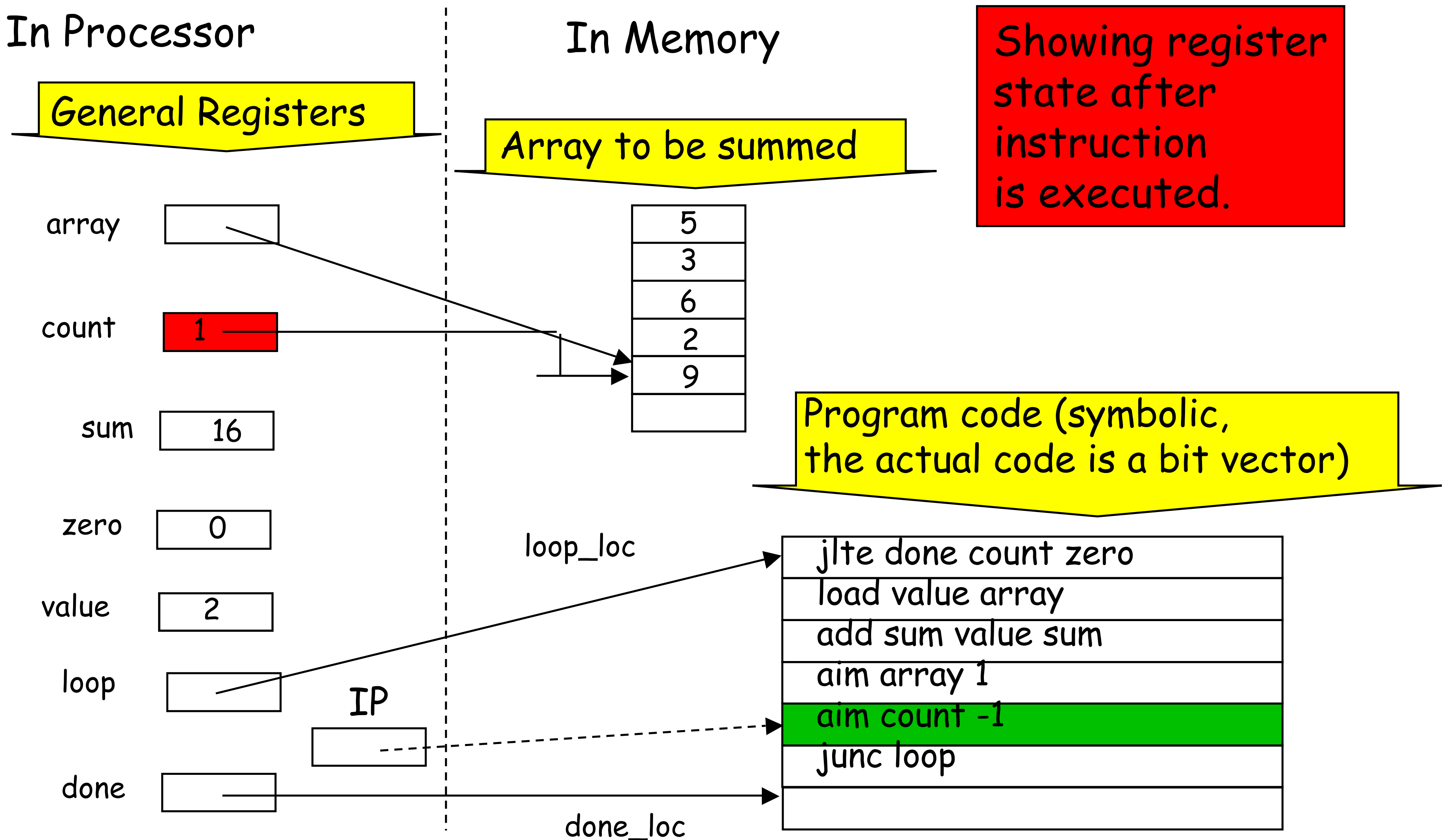
Array Summation



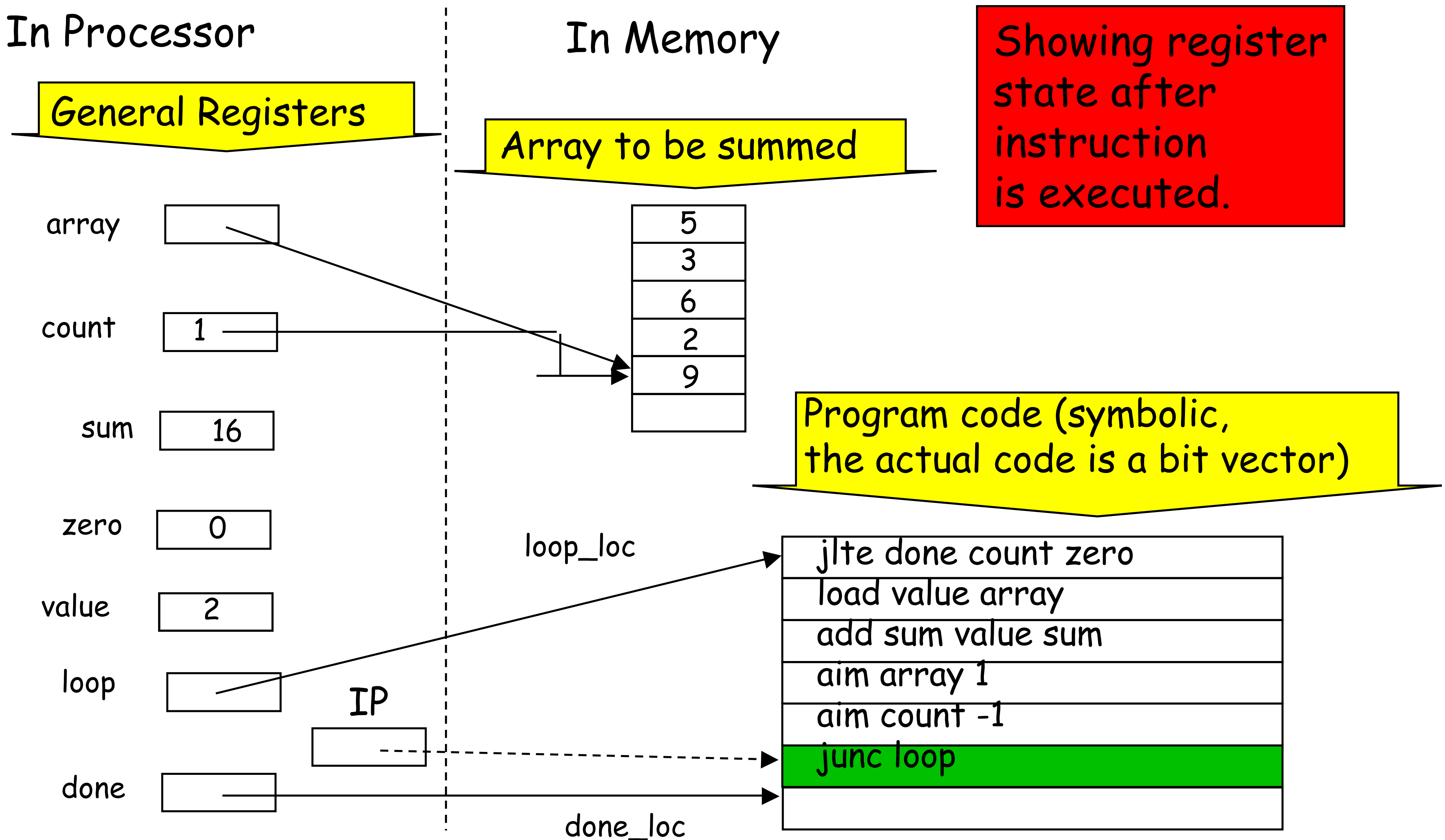
Array Summation



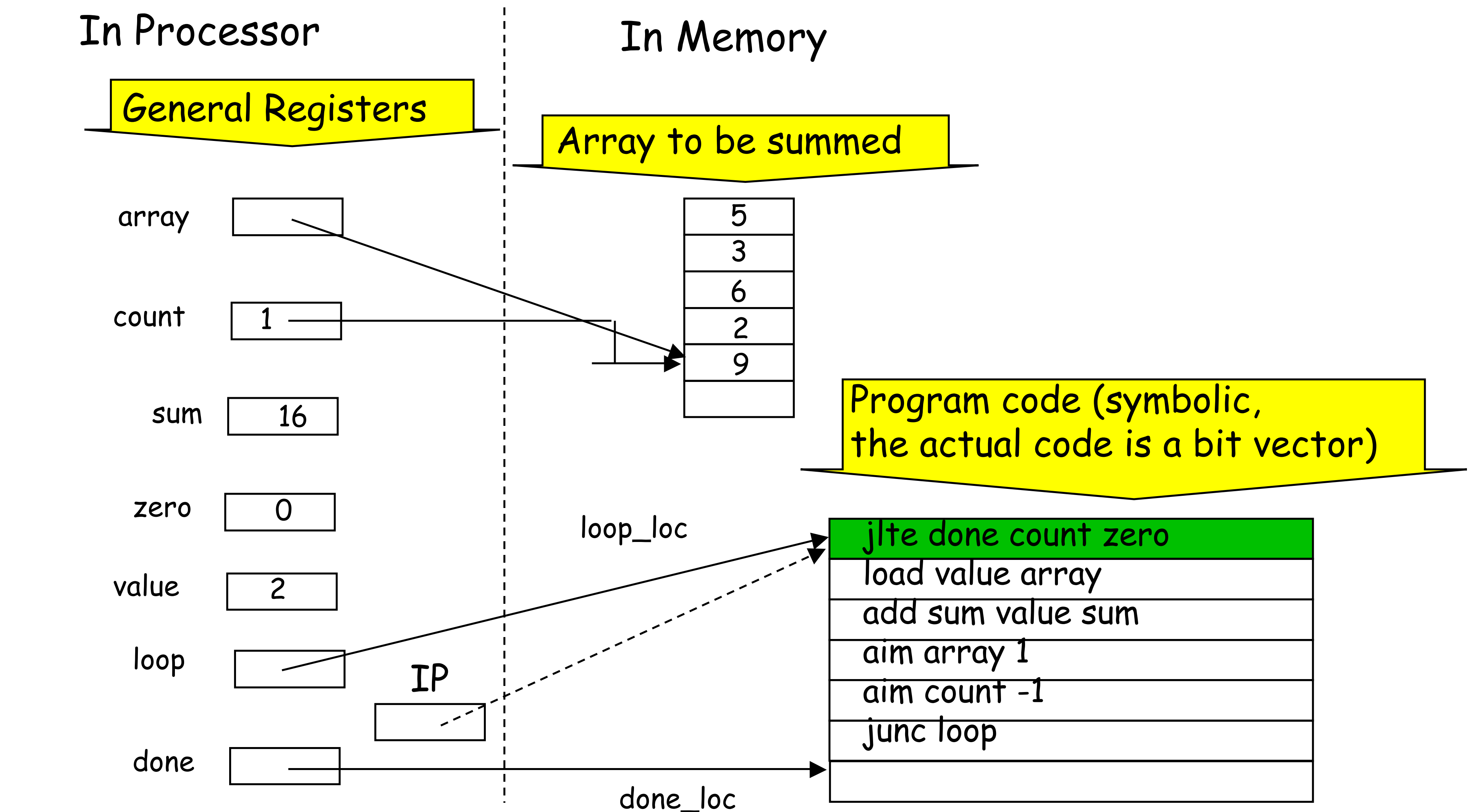
Array Summation



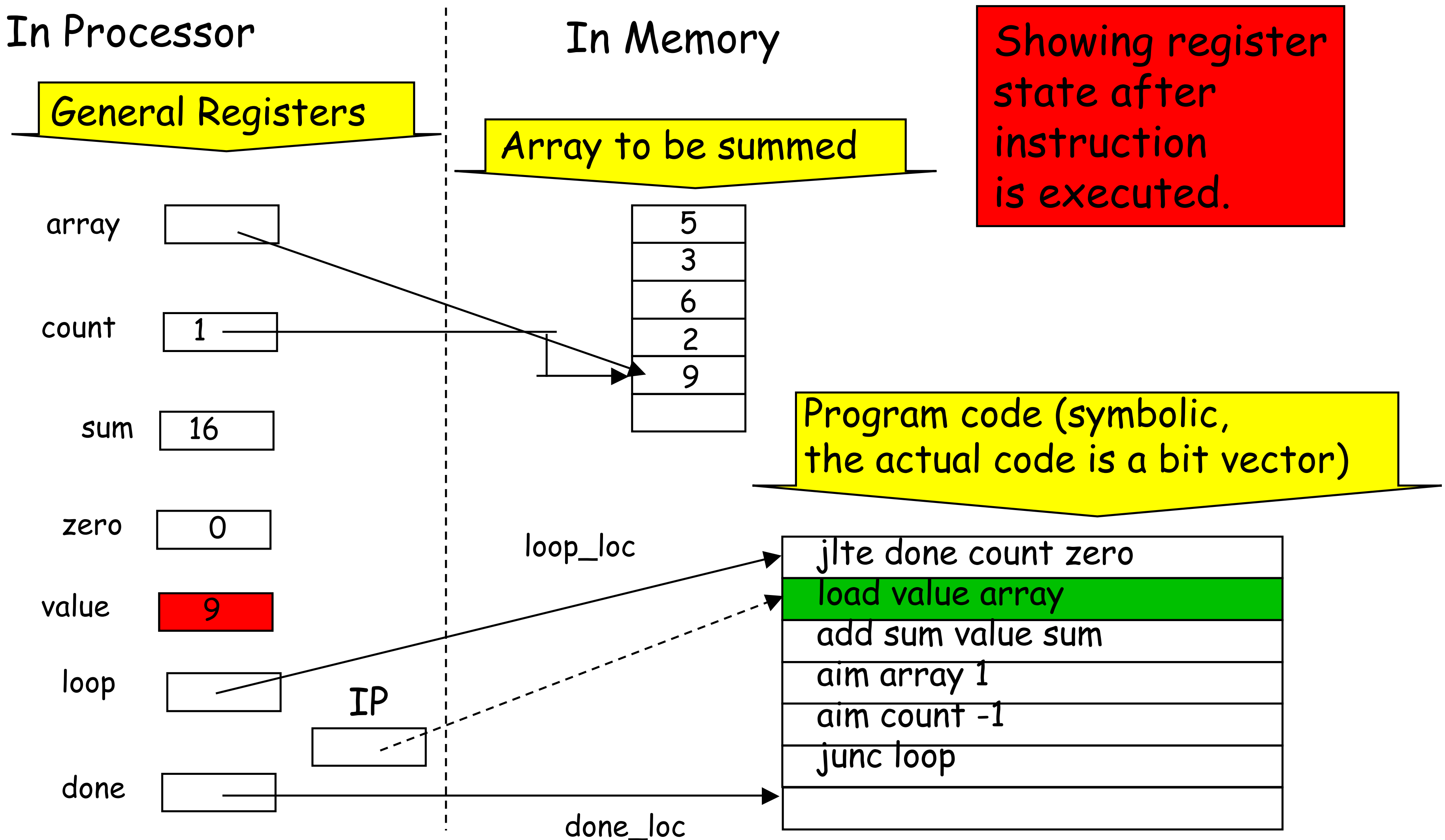
Array Summation



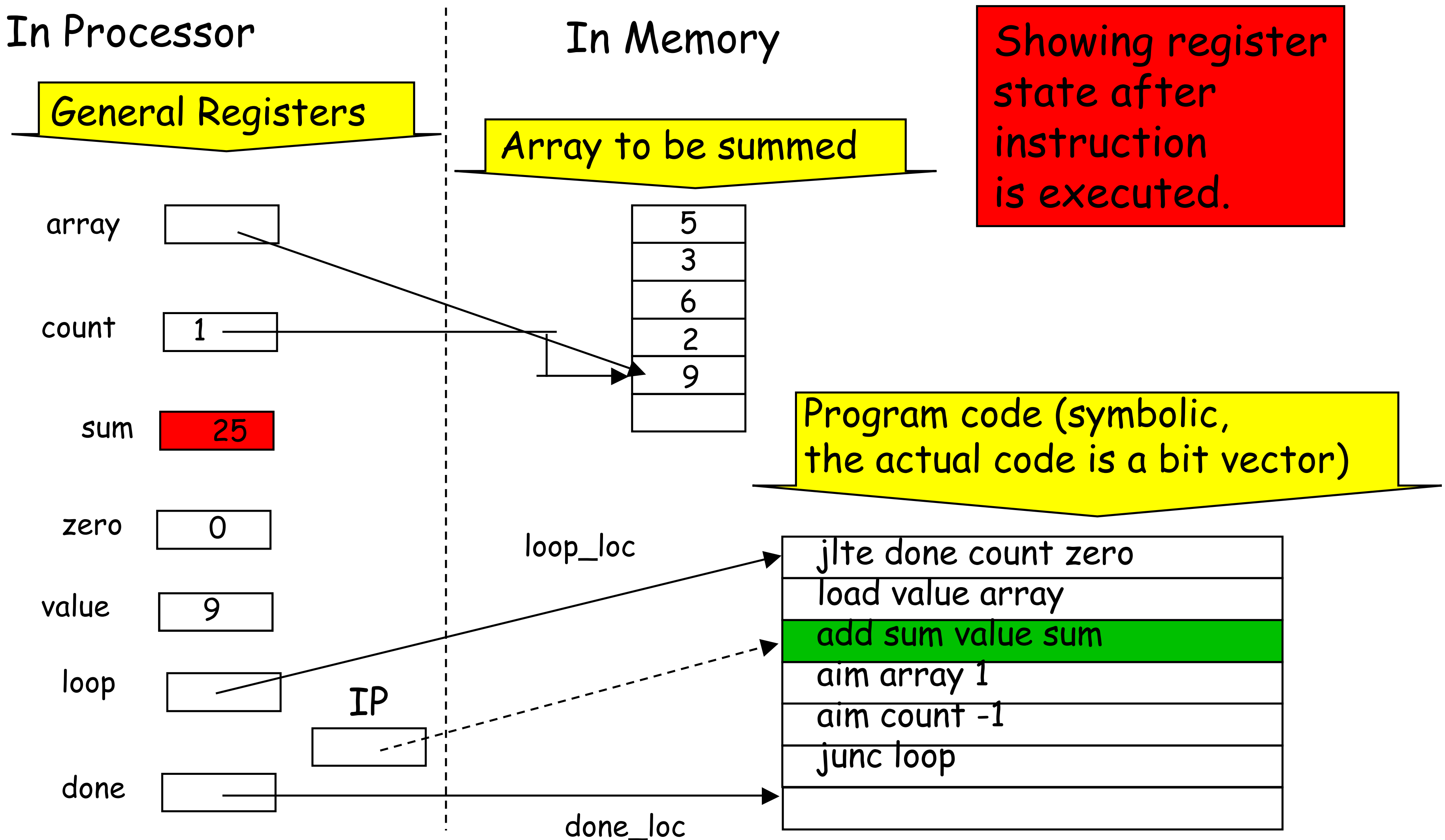
Array Summation



Array Summation

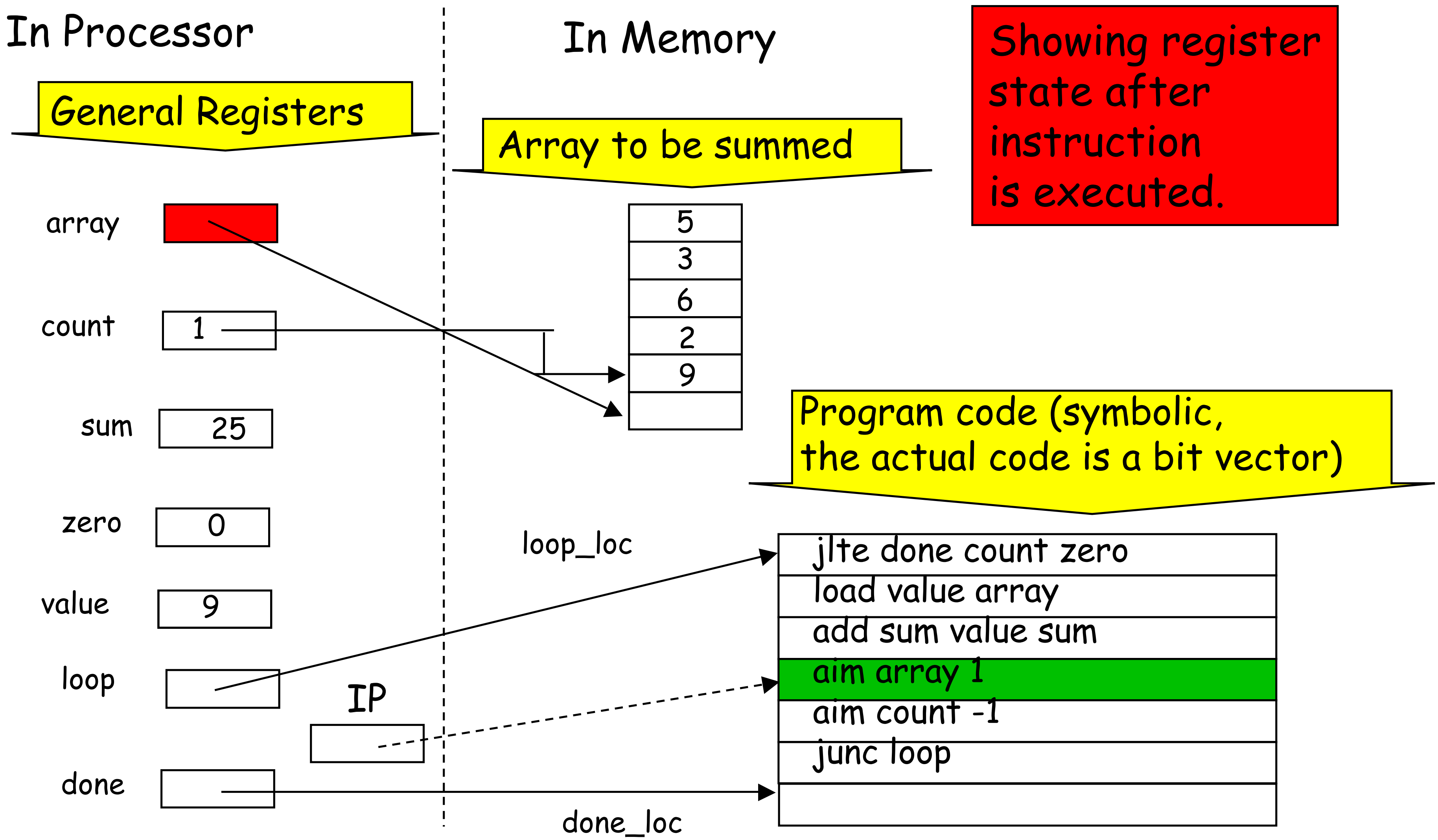


Array Summation

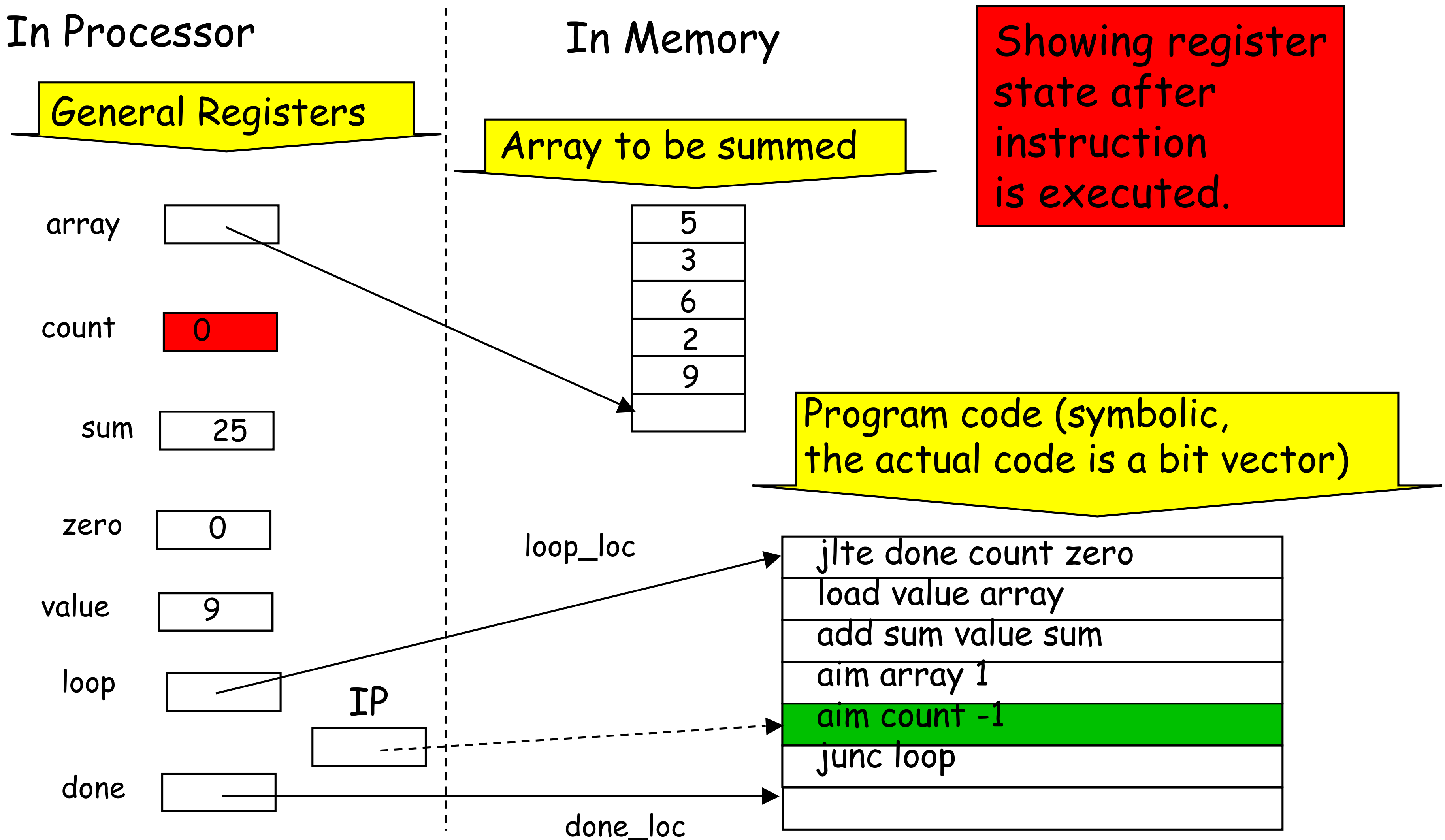


Showing register state after instruction is executed.

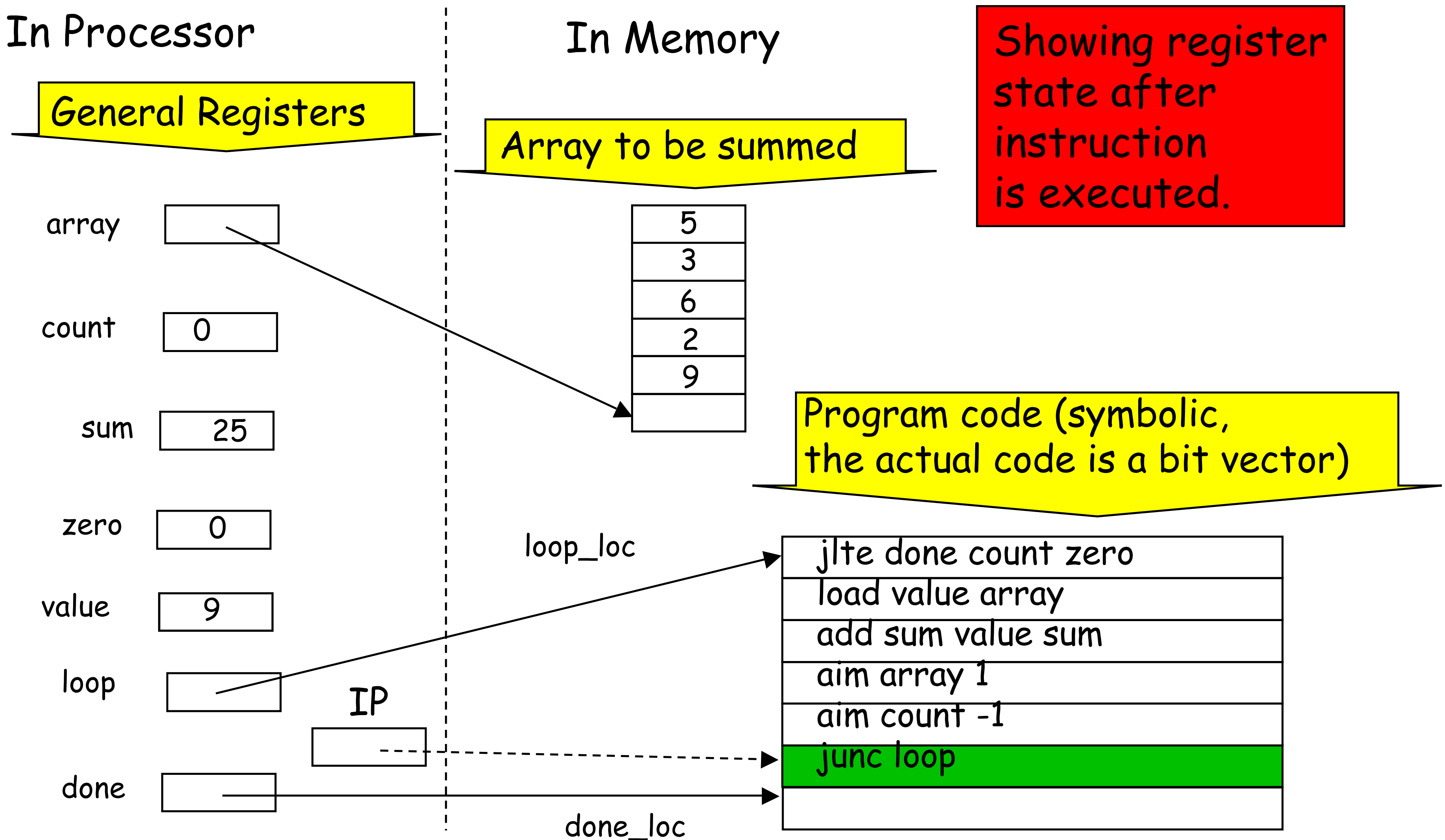
Array Summation



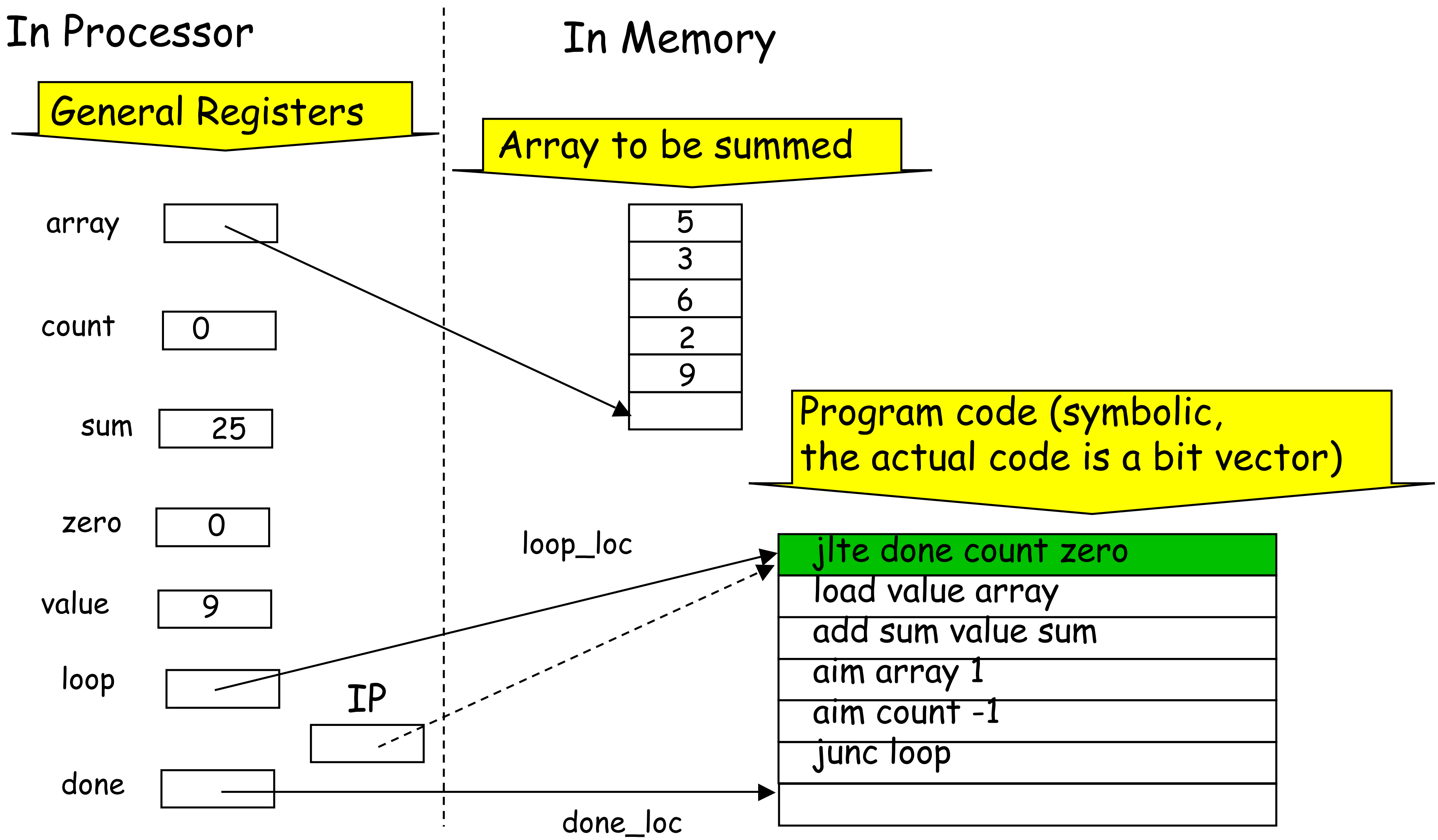
Array Summation



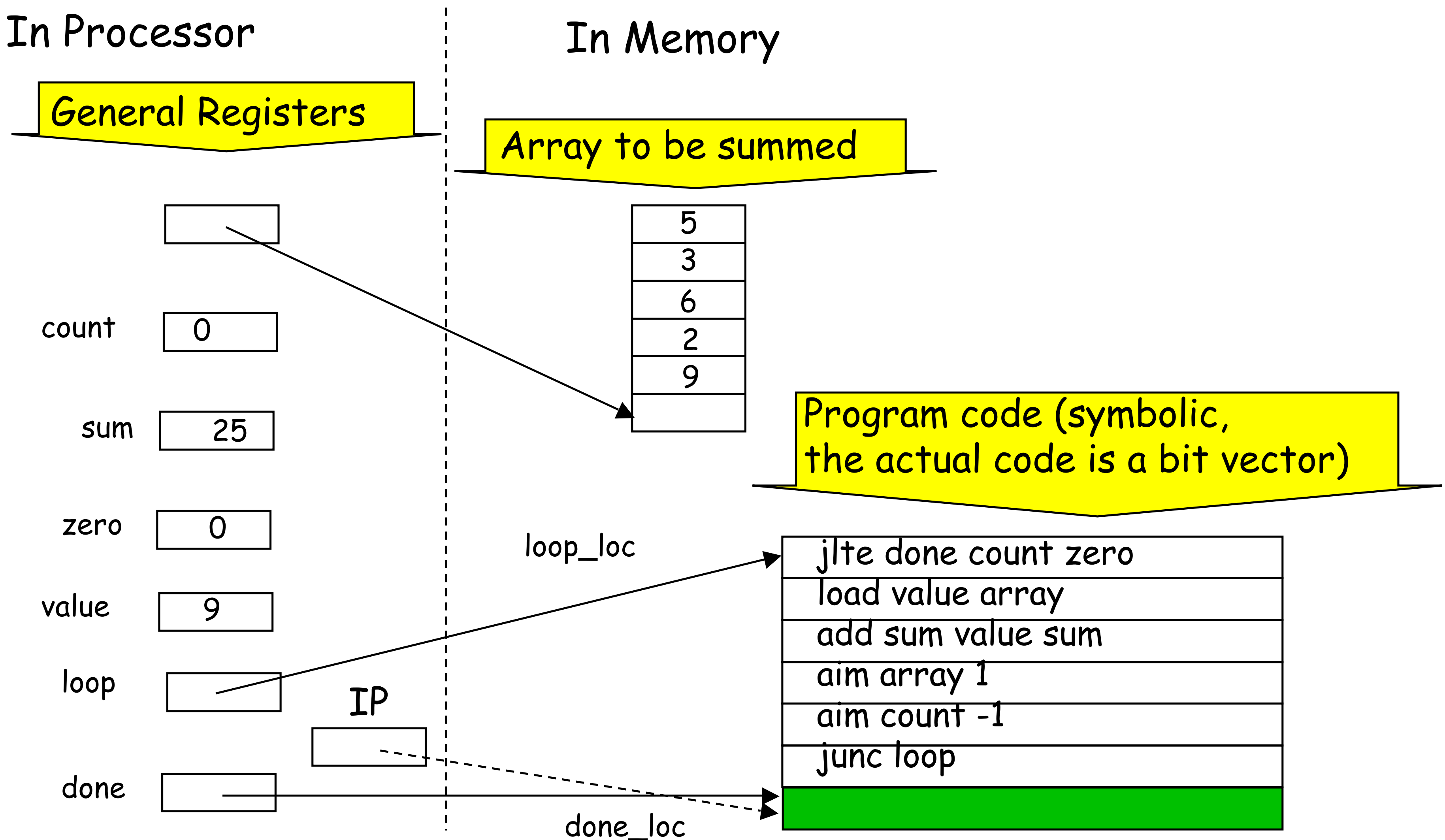
Array Summation



Array Summation



Array Summation



The ISC Assembler

- The ISC assembler is actually a combined assembler, loader, and tracer (for debugging).
- The executable is: /cs/cs60/bin/isc
- Sample programs are in /cs/cs60/isc/

The ISC Assembler

- The array summation program is in /cs/cs60/isc/array.isc. It includes additional code to create the array and output the result.

isc output

```
turing isc:1> isc array.isc
25

turing isc:2> isc -t array.isc
            Begin trace
line:   36 loc:    0 contents: lim     r9  (0)            0
    reg[9] = 0

. . . lots of trace output . . .
------- output ------
25
--------------------
line:   88 loc:   32 contents: junc    r9  (0)
    jumping unconditionally to 0
```

trace flag

Implementing Procedures

- A **procedure** is just some fixed code we jump to to perform some computation.
- We put the **arguments** to the procedure into pre-agreed **standard registers**, and get the result from another register.
- The **jsub** instruction can be used to get the **return address**.

Implementing Recursion

- Recursion presents a problem: The fixed code would tend to clobber (an inelegant way of saying “over-write”) the return address when the procedure calls itself.
- Also, the arguments in registered would get clobbered.

Stacks to the Rescue

- To deal with the clobbering problem, we can **save** the return address and any arguments on a **stack** allocated at a standard place in memory.
- The stack is typically an array, with a pointer to the top element.

```
lim stack_pointer save_area_loc  // initialize stack pointer

aim stack_pointer    -1          // always point to top of stack
```

Example: Recursive Factorial

```
    label fac                          // recursive factorial routine
        lim result 1                   // basis is 1
        jlte return arg zero           // return if count is 0 or less

push    aim  stack_pointer +1          // increment stack pointer
        store stack_pointer return     // save return address on stack

push    aim  stack_pointer +1          // increment stack pointer
        store stack_pointer arg        // save argument on stack

        aim arg -1                     // subtract 1 from argument
        jsub jump_target return        // call recursively

pop     load arg stack_pointer         // restore original argument
        aim  stack_pointer -1

pop     load return stack_pointer      // restore original return address
        aim  stack_pointer -1

        mul  result result arg         // multiply by original arg
        junc return                    // return to caller
```

Stack trace of factorial(4)

Stack trace of factorial(4)