05- Control-flow & Functions in AVR

Ahmad Abdullah, PhD
abdullah@uvic.ca
https://web.uvic.ca/~abdullah/csc230

Lectures: MR 10:00 – 11:20 am
Location: ECS 125

Outline

- Review of control-flow instructions so far
- Function calls (simple)
- The runtime stack
- Return values, parameters
- Call-by-value vs. call-by-reference
- Registers vs. stack for parameters
- Stack frames

Control flow so far

- Unconditional relative jump instruction:
  - RJMP <program label>
- Conditional branch instruction:
  - BR<some condition> <program label>
  - AVR specifies 20 such instructions

```
    ; some code ...
    ldi r16, 1
    ldi r17, 0
loop12times:
    add r17, r16
    cpi r17, 12
    brne loop12times ; assembler converts "loop12times" to an address
    ; some code ...
```

```
    ; some code ...
stop:
    rjmp stop              ; assembler coverts "stop" to address
```

Translating “if” structures

- Note:
  - Many, many more control-flow possibilities are possible in assembler...
  - ... yet you understand the Java “if” statement ...
  - ... and may want to see some AVR translations
- This will also help demonstrate more branch instructions
- Assume all operands are one-byte unsigned integers
- Also assume .def directive used to map a name to a register

```
.cseg
; ... some lines
.def a=r16
.def b=r17
.def m=r18
.def n=r19
```

Translating “if” structures

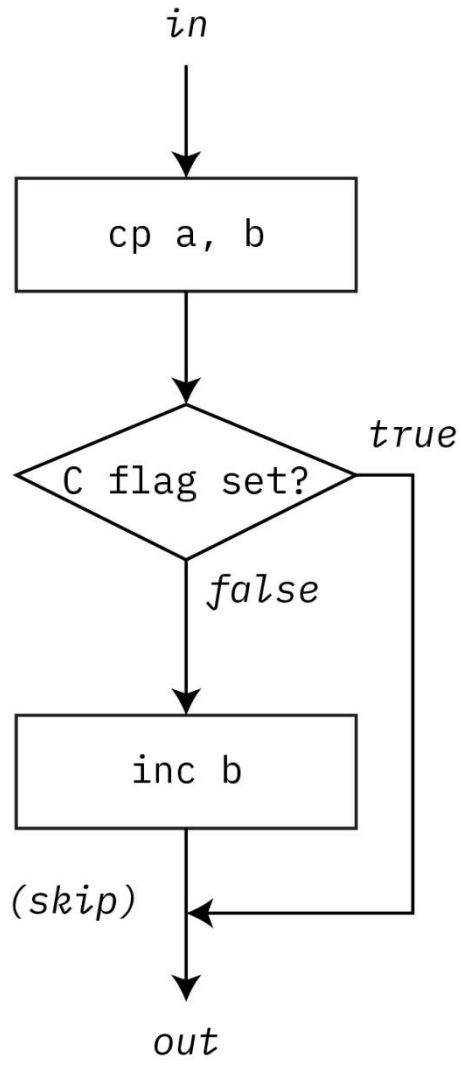
```
if (a >= b) {
    b++;
}
```

```
    cp a, b
    brlo skip ; skip if a < b
    inc b
skip:
```

brlo: "branch if low" (branch to destination if carry flag is set)

Note the inverted logic that we use for assembly:

!(a < b) ←→ a >= b



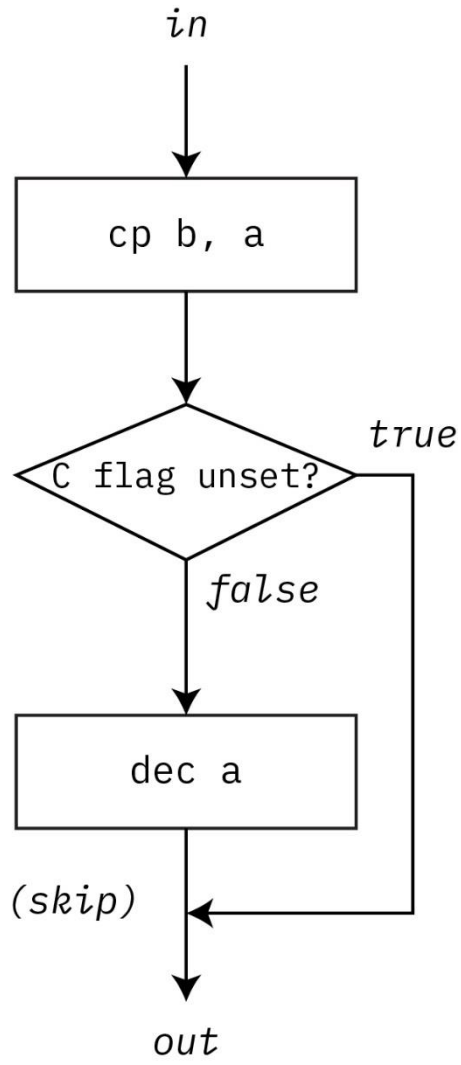
Translating “if” structures (Cont.)

```
if (a > b) {
    a--;
}
```

```
    cp a, b
    brsh skip ;skip if a <= b
    dec b
skip:
```

brsh: "branch if same or higher" (branch to destination if carry flag is cleared / not set)

It may help if you understand that "cp Rd, Rr" is the same as computing Rd – Rr and setting SREG flags but not saving the result of the subtraction.



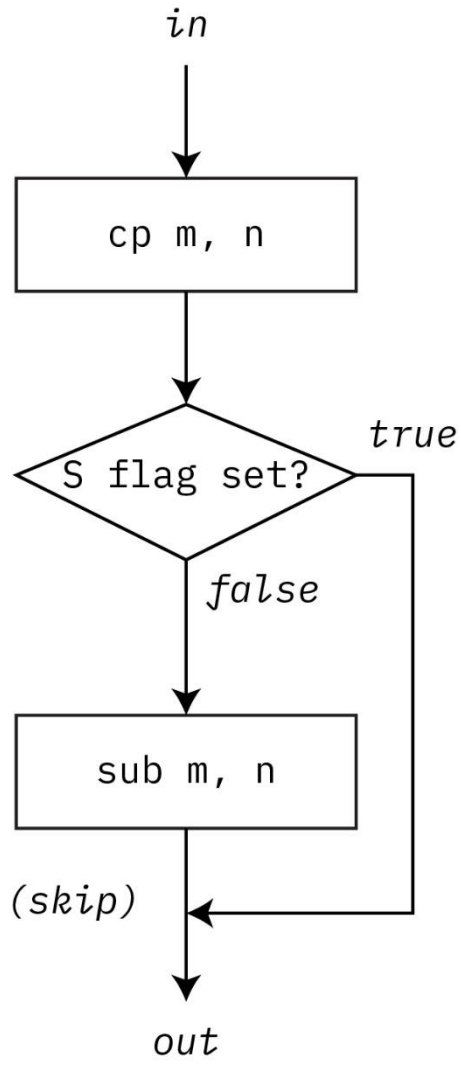
Translating “if” structures (Cont.)

```
if (m >= 0) {
    m -= n;
}
```

```
    cp m, n
    brlt skip ;skip if m < n
    sub m, n
skip:
```

brlt: "branch if less than (signed)"; if the Signed flag is set, then branch is taken.

Note similarity with first example...



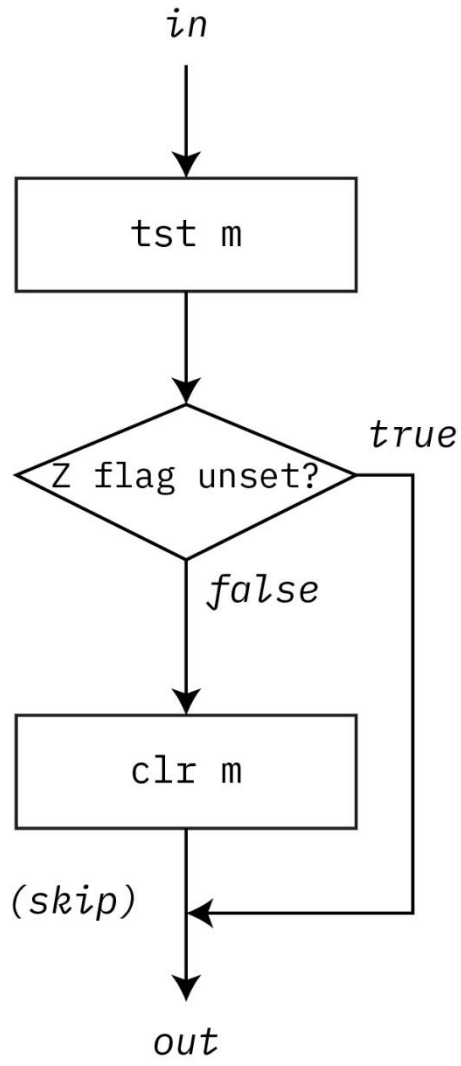
Translating “if” structures (Cont.)

```
if (m == 0) {
    n = 0;
}
```

```
    tst m
    brne skip ;skip if m <> 0
    clr n
skip:
```

tst Rd: "test if register is zero or negative"; Z flag is set or cleared depending on value in Rd

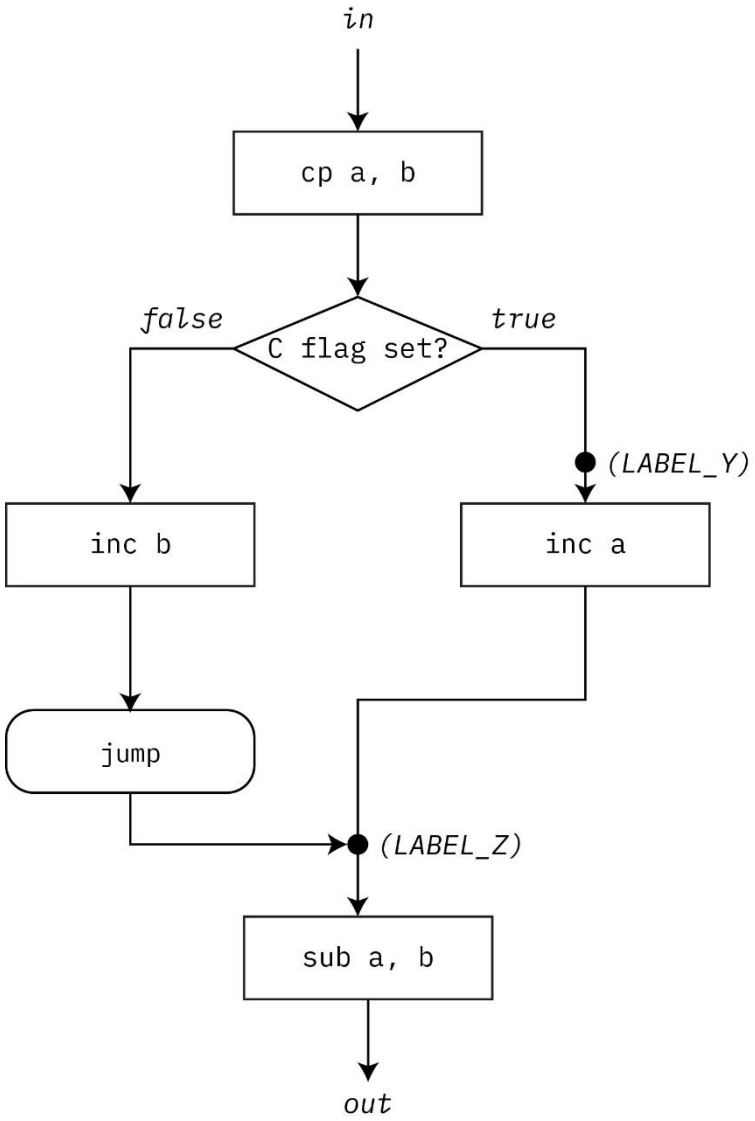
clr Rd: "clears register Rd"



Translating an “if-else” structure

```
if (a >= b) {
    b++;
} else {
    a++;
}
a -= b;
```

```
    cp a, b
    brlo LABEL_Y
    inc b
    rjmp LABEL_Z
LABEL_Y:
    inc a
LABEL_Z:
    sub a, b
```



JMP

- JMP = (unconditional) jump

JMP – Jump

Description:

Jump to an address within the entire 4M (words) Program memory. See also RJMP.

This instruction is not available in all devices. Refer to the device specific instruction set summary.

Operation:

(i) PC ← k

	Syntax:	Operands:	Program Counter:	Stack:
(i)	JMP k	0 ≤ k < 4M	PC ← k	Unchanged

32-bit Opcode:

1001	010k	kkkk	110k
kkkk	kkkk	kkkk	kkkk

JMP (Cont.)

- Notice the range of addresses possible
  - Requirement is 0 <= k < 4M
  - Therefore k ranges from 0 to 2^{22} -1
- This address k is **absolute**
  - It refers to a location in program memory
  - Also this range is far, far larger than what is needed for the mega2560. (Why?)
- Note the way the opcode is encoded

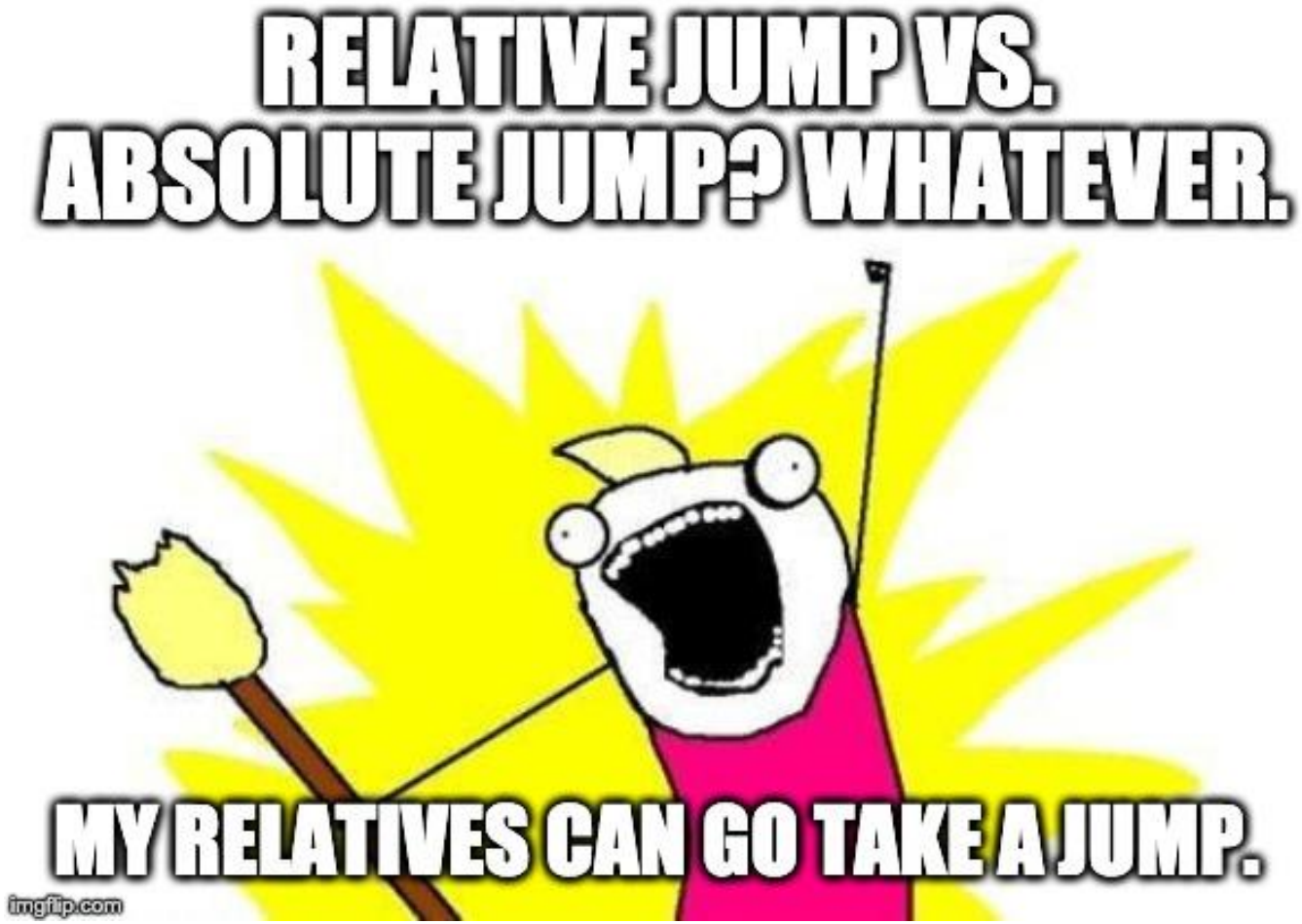
	Syntax:	**Operands:**
(i)	JMP k	$0 \le k < 4M$

32-bit Opcode:

1001	010k	kkkk	110k
kkkk	kkkk	kkkk	kkkk

"But haven't we already stated that program-memory words are addressed by 17-bit addresses? Now we see 22 bits. This is confusing..."

But.... why not just JMP?



But.... why not just JMP?

- We need both formats
- RJMP:
  - Encoded opcode **is two bytes** in size
  - Execution requires **two** cycles
- JMP:
  - Encoded opcode is **four bytes** in size
  - Execution requires **four** cycles
- Given the AVR architecture is meant for embedded systems:
  - RJMP requires less CPU and memory, and helps when memory available is very limited
  - In fact, some AVR implementations do not have JMP implemented (i.e., flash memory is too small).

And remember...

Program Memory

AVR architecture supports program memories up to 4M **words** in size

Actual program memory depends upon size of implemented chip's PC (from 16-bits to 22-bits)

mega2560 has a 17-bit PC: **128K words**

≠

Data Memory

May be up to 64K **bytes** in size

Contains with that range of 64K address the SRAM for use with program data (mega2560: **8K bytes in SRAM**)

Control flow beyond branches

- In higher-level languages we often use this very important form of control flow:
  - **call** some “function” f() ...
  - ... which means **computer transfers control** to the first instruction in f()
  - ... then have the computer **execute the code corresponding to function f()** ...
  - ... and when the code in this function is completed, **control resumes at the instruction right after the original call** to f()
- Function-call semantics are very rich...
  - ... input parameters ...
  - ... return types ...
  - but we’ll initially ignore these in AVR

Example

```
.cseg
.org 0

    ldi r16, (1591 & 0xff)
    ldi r17, ((1591 >> 8) & 0xff)
    ldi r18, (312 & 0xff)
    ldi r19, ((312 >> 8) & 0xff)

    ldi r20, 9
loopA:
    add r16, r18
    adc r17, r19
    dec r20
    brne loopA

    ; .. do something with result
    ; in r17:r16
    ;...

    ldi r16, (209 && 0xff)
    ldi r17, ((209 >> 8) && 0xff)
    ldi r18, (11 && 0xff)
    ldi r19, ((11 >> 8) && 0xff)

    ldi r20, 9
loopB:
    add r16, r18
    adc r17, r19
    dec r20
    brne loopB

    ; .. do something with result
    ; in r17:r16
    ; ...

done:
    jmp done
```

Imagine we want to add r19:r18 to r17:r16 nine times...

... and want to do this in several places.

Example (Cont.)

```
.cseg
.org 0

    ldi r16, (1591 & 0xff)
    ldi r17, ((1591 >> 8) & 0xff)
    ldi r18, (312 & 0xff)
    ldi r19, ((312 >> 8) & 0xff)
    call add_nine_times

    ; ... do something with result
    ; in r17:r16

    ldi r16, (209 && 0xff)
    ldi r17, ((209 >> 8) && 0xff)
    ldi r18, (11 && 0xff)
    ldi r19, ((11 >> 8) && 0xff)
    call add_nine_times

    ; .. do something with result
    ; in r17:r16

done:
    jmp done
```

```
add_nine_times:
    ldi r20, 9
loop:
    add r16, r18
    adc r17, r19
    dec r20
    brne loop
    ret
```

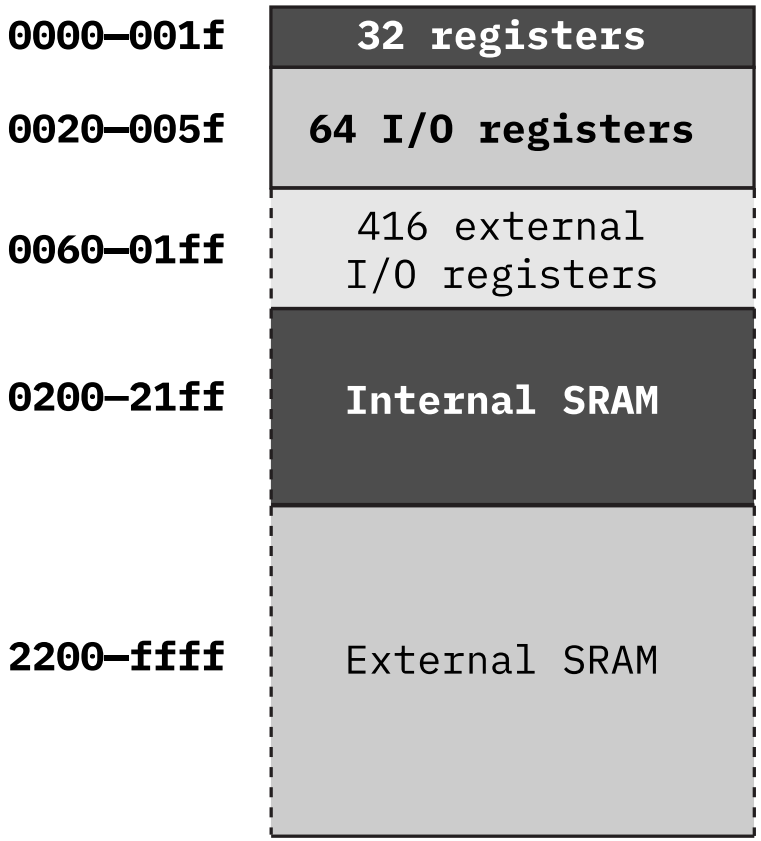
Previous example, but now with the addition loop placed into a function.

CALL and RET

- Before branching to a new program-memory location indicated by the CALL instruction:
  - ... the processor saves the current PC value onto the **stack** ...
  - ... and this value is the **program-memory location following the CALL instruction itself.**
- The address saved on the stack is called the **return address**
- After the code in the called function is complete...
  - ... the RET instruction (for “return”) is executed ...
  - ... which **overwrites the PC with the most recent values** saved on the stack
- Both CALL and RET **modify the PC**...
  - ... and therefore modify the normal sequential flow-of-control

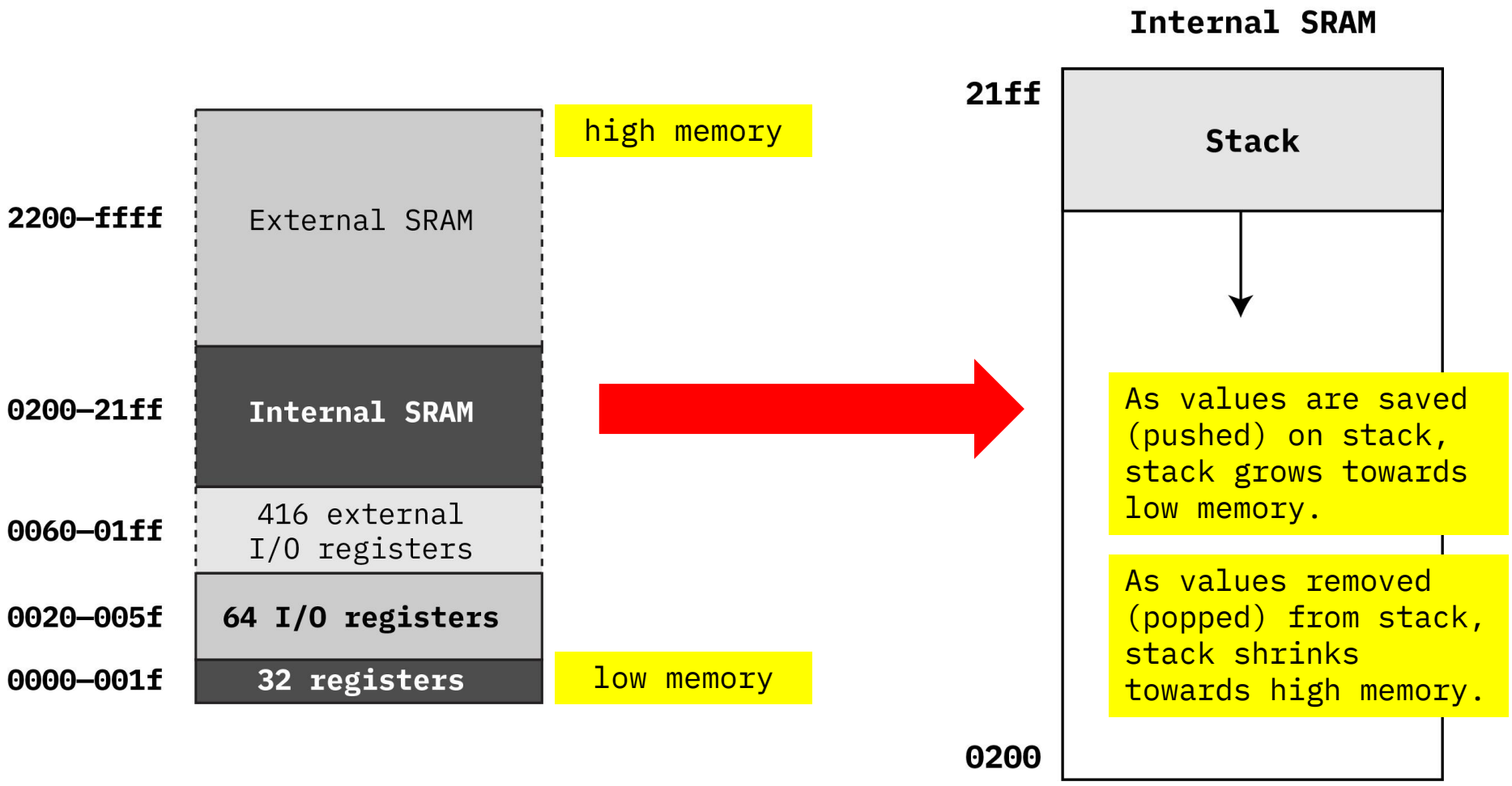
But what is / where is this “stack” of which we speak?

Recall this diagram



In describing computer systems and memory, however, we normally turn this diagram upside down.

Data memory


2200–ffff
External SRAM
high memory
0200–21ff
Internal SRAM
0060–01ff
416 external I/O registers
0020–005f
64 I/O registers
0000–001f
32 registers
low memory
Internal SRAM
21ff
Stack
As values are saved (pushed) on stack, stack grows towards low memory.
As values removed (popped) from stack, stack shrinks towards high memory.
0200

Stacks

- You have seen (and perhaps coded the operations for) general-purpose stacks in CSC 115/116.
  - last-in, first-out storage structure
- Most processor designs include **hardware support** for **operations** on **at least one stack**
- The AVR processor includes a **stack pointer (SP)** register ...
  - This register is actually **stored in the I/O register area**
- ... and also includes some **explicit stack operations** (POP, PUSH)
  - However, for CALL and RET, the stack operations are **implicit**

Implicit operations

- Before we look at explicit stack operations (POP and PUSH) ...
- ... let's dig deeper into what is involved with the stack in CALL and RET operations
  - ... **especially use of the SP (stack pointer)**
- Remember:
  - Each data memory address refers to **one byte** of data memory storage
  - Each program-memory address accesses **one word** of program memory storage
  - The program-memory address itself, however, is 17 bits wide.
  - To save a program-memory address on the mega2560 means **to save its 17 bits** (i.e., three bytes)

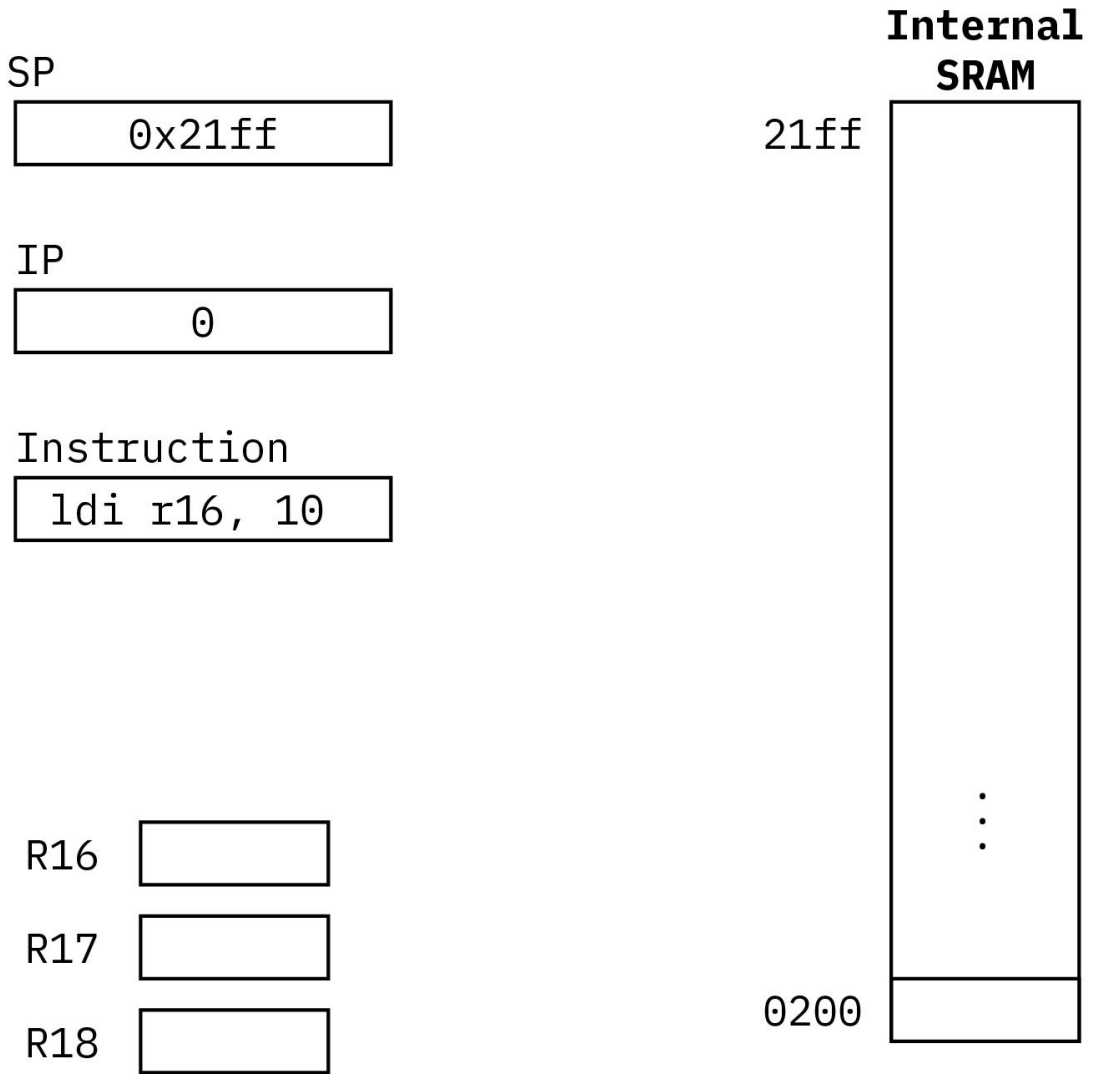
Example

```
cseg
.org 0  ;

    ldi r16, 10
    ldi r17, 5

    call foo

    ldi r18, 1
    add r17, r18

    call foo

stop:
    rjmp stop

foo:
    add r17, r16
    ret
```

```
Hex from assembled opcodes,
Word addresses, word contents big-endian.

0000 E00A
0001 E015

0002 940E0009

0004 E021
0005 0F12

0006 940E0009


0008 CFFF


0009 0F10
0010 9508
```

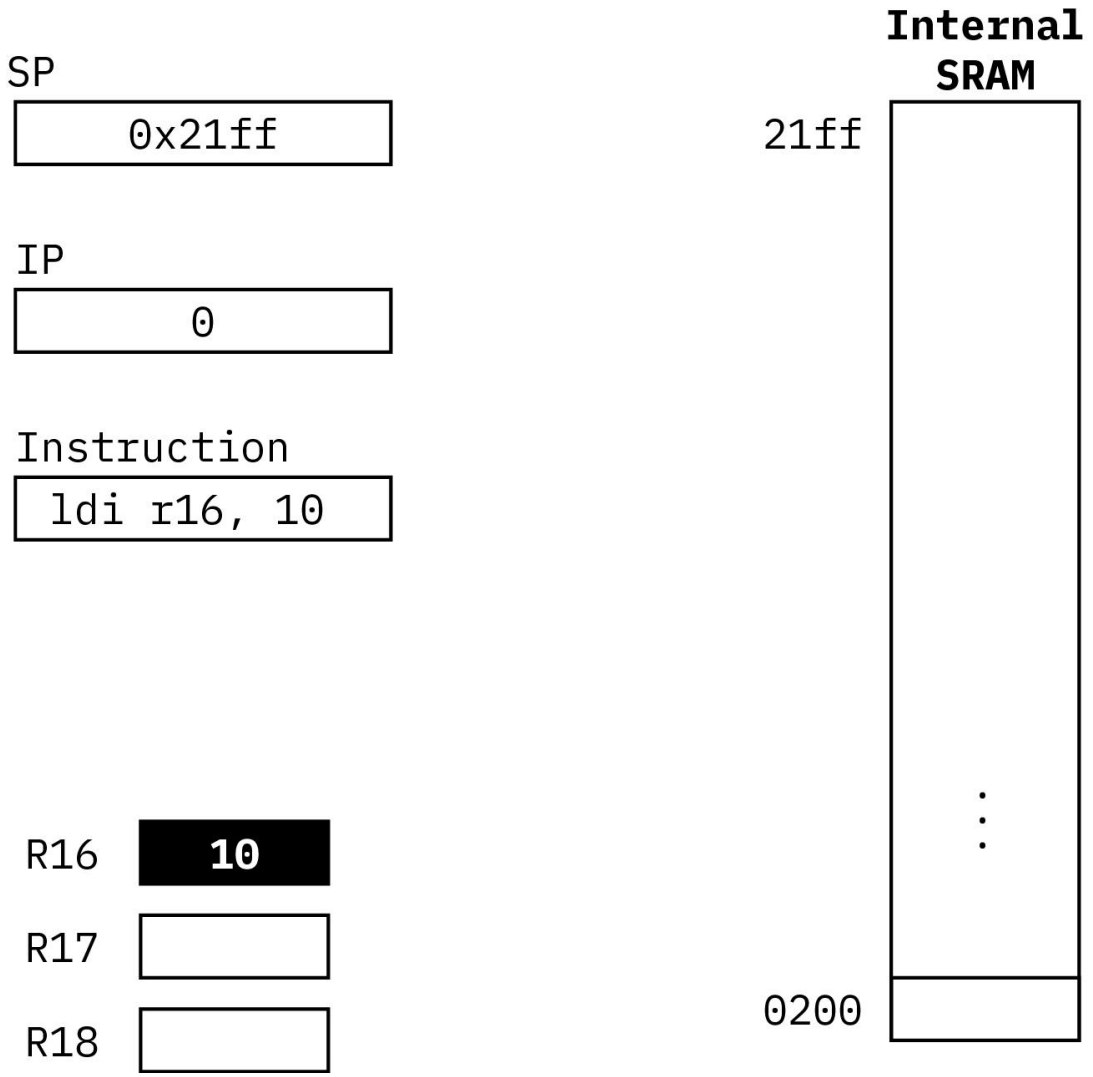
Start cycle

```
.org 0
    ldi r16, 10
    ldi r17, 5
    call foo
    ldi r18, 1
    add r17, r18
    call foo

stop:
    rjmp stop

foo:
    add r17, r16
    ret
```



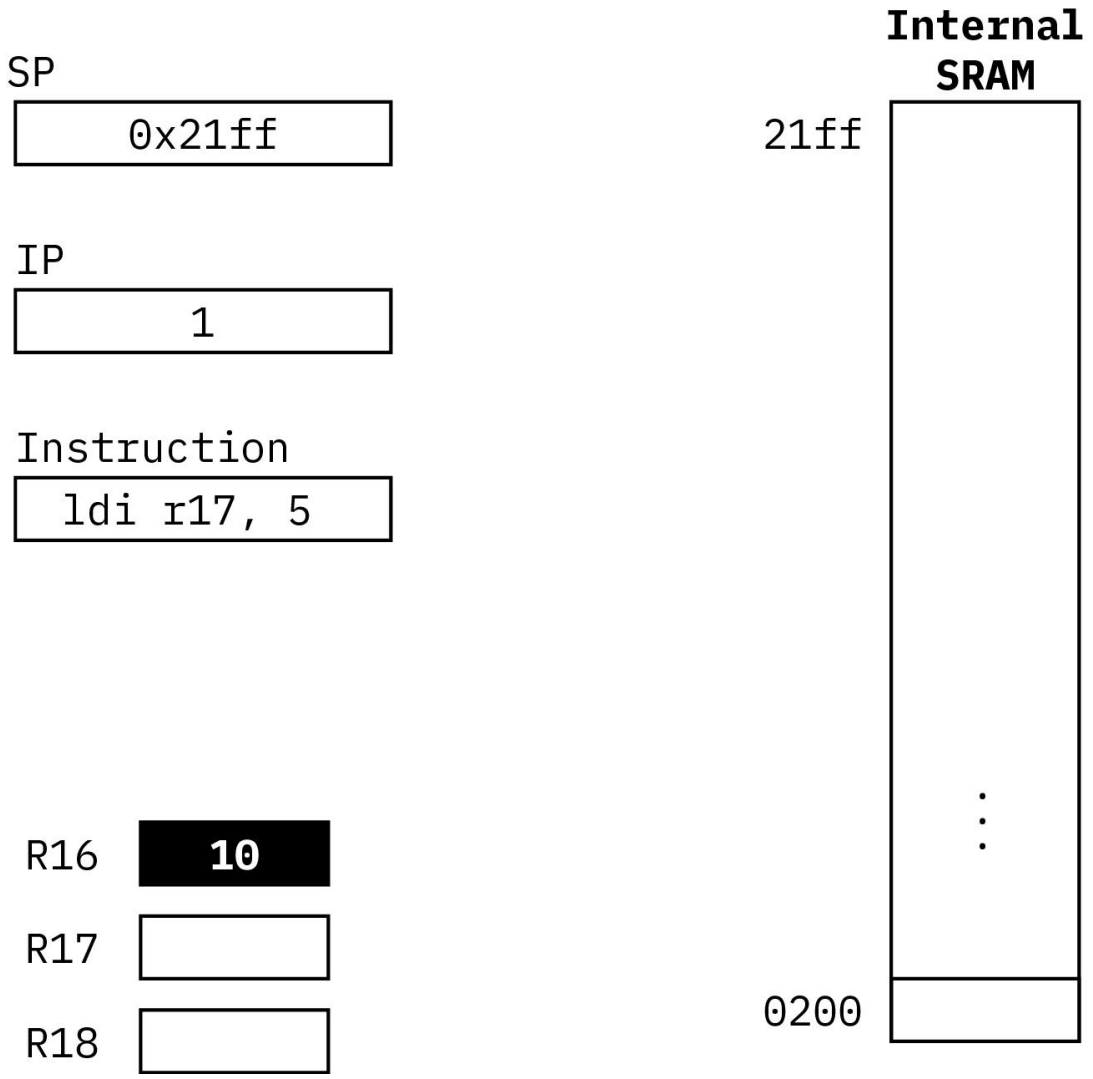
Finish cycle

```
.org 0
    ldi r16, 10
    ldi r17, 5
    call foo
    ldi r18, 1
    add r17, r18
    call foo

stop:
    rjmp stop

foo:
    add r17, r16
    ret
```



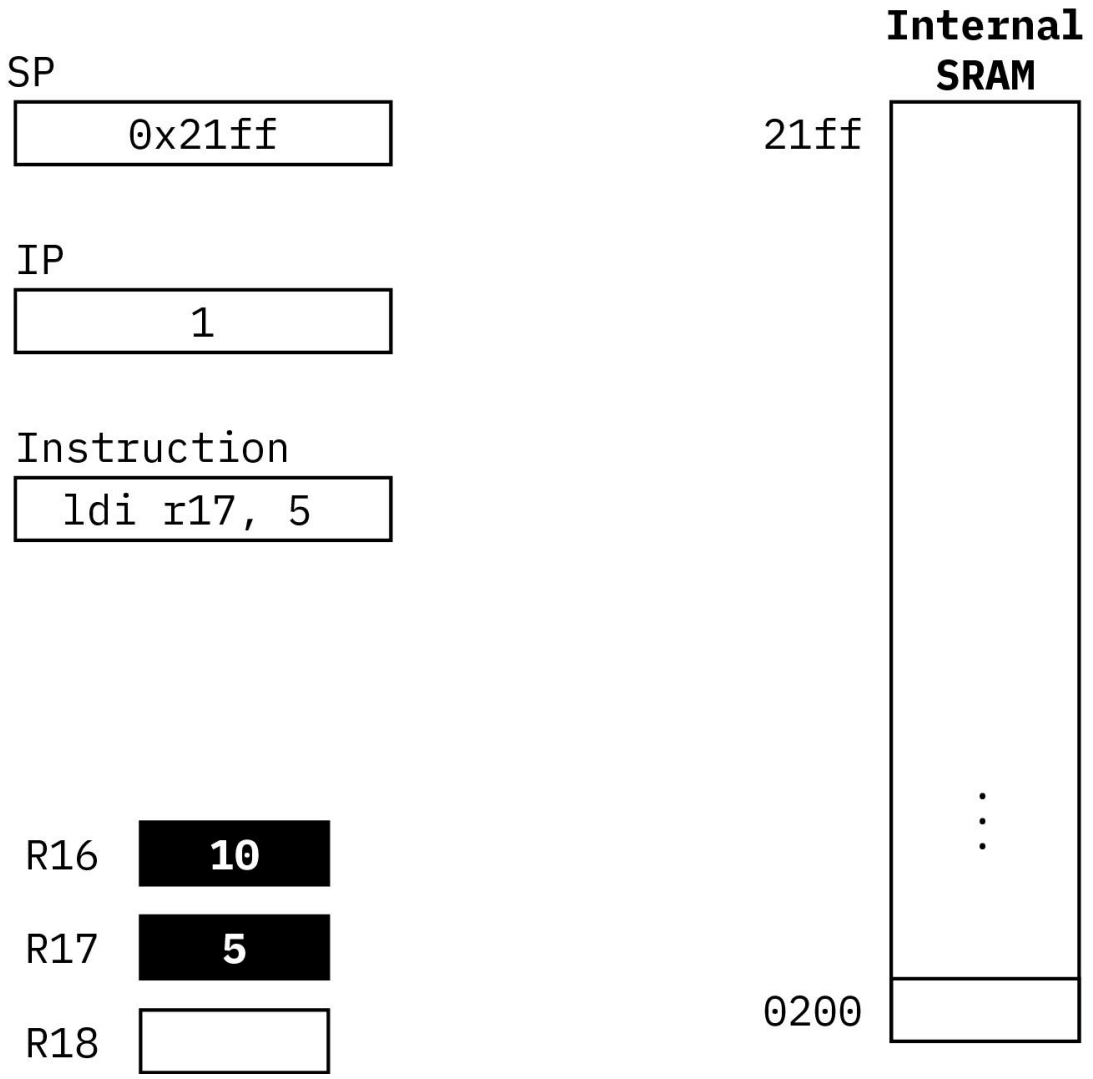
Start cycle

```
.org 0
    ldi r16, 10
    ldi r17, 5
    call foo
    ldi r18, 1
    add r17, r18
    call foo

stop:
    rjmp stop

foo:
    add r17, r16
    ret
```



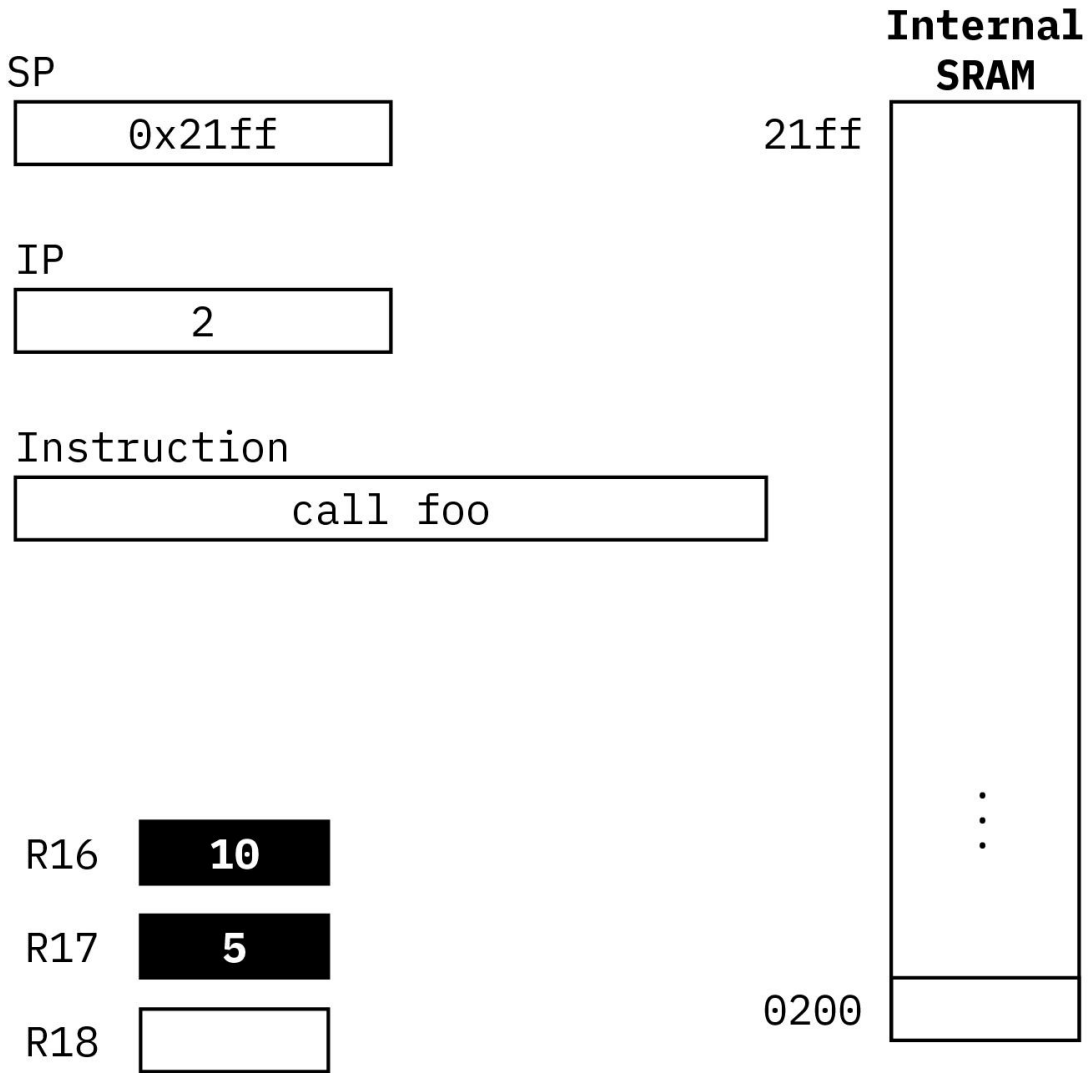
Finish cycle

```
.org 0
    ldi r16, 10
    ldi r17, 5
    call foo
    ldi r18, 1
    add r17, r18
    call foo

stop:
    rjmp stop

foo:
    add r17, r16
    ret
```



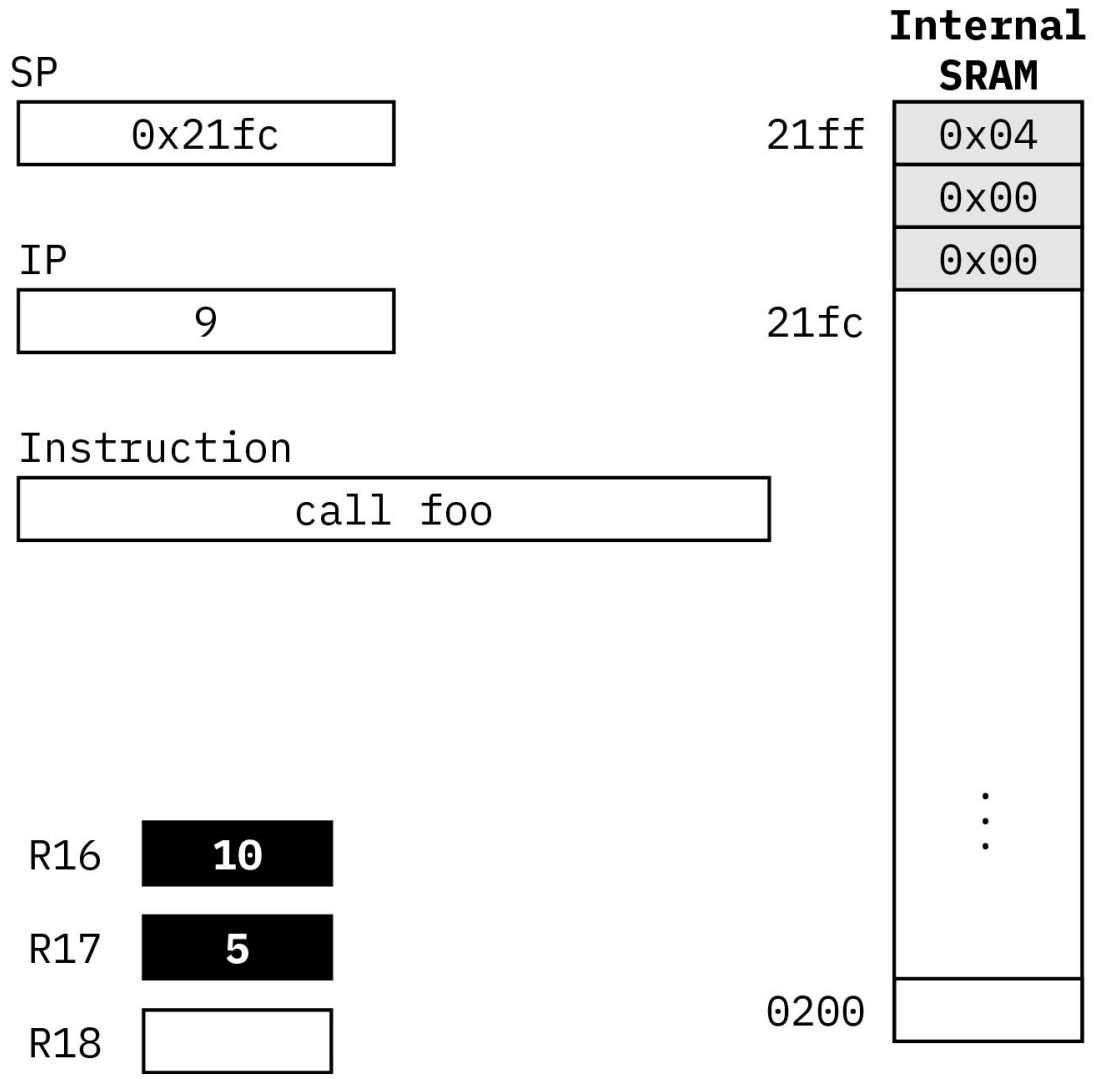
Start cycle

```
.org 0
    ldi r16, 10
    ldi r17, 5
    call foo
    ldi r18, 1
    add r17, r18
    call foo

stop:
    rjmp stop

foo:
    add r17, r16
    ret
```



Finish cycle

```
.org 0
    ldi r16, 10
    ldi r17, 5
    call foo
    ldi r18, 1
    add r17, r18
    call foo

stop:
    rjmp stop

foo:
    add r17, r16
    ret
```

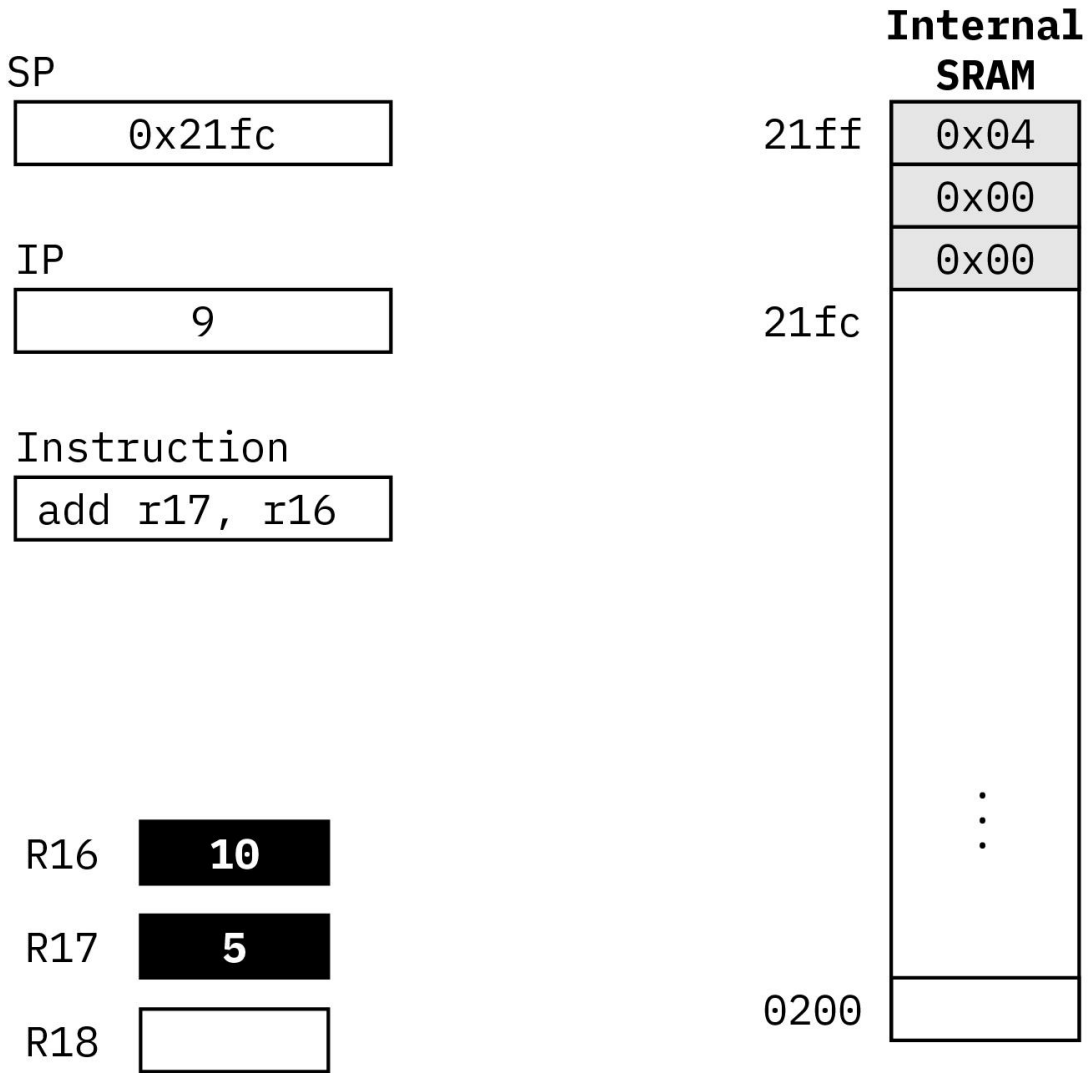


Short interruption

- Notice the order in which bytes appear on the stack.
  - The address 0x00004 was pushed on the stack.
  - Which means 0x04 was pushed first...
  - ... then the first 0x00 ...
  - ... and then the last 0x00 ...
  - ... which covers the 17 bits of the PC
- These pushes are implicitly performed by the CPU
  - We do not need to inform the CPU to push the addresses
  - CALL does this for us

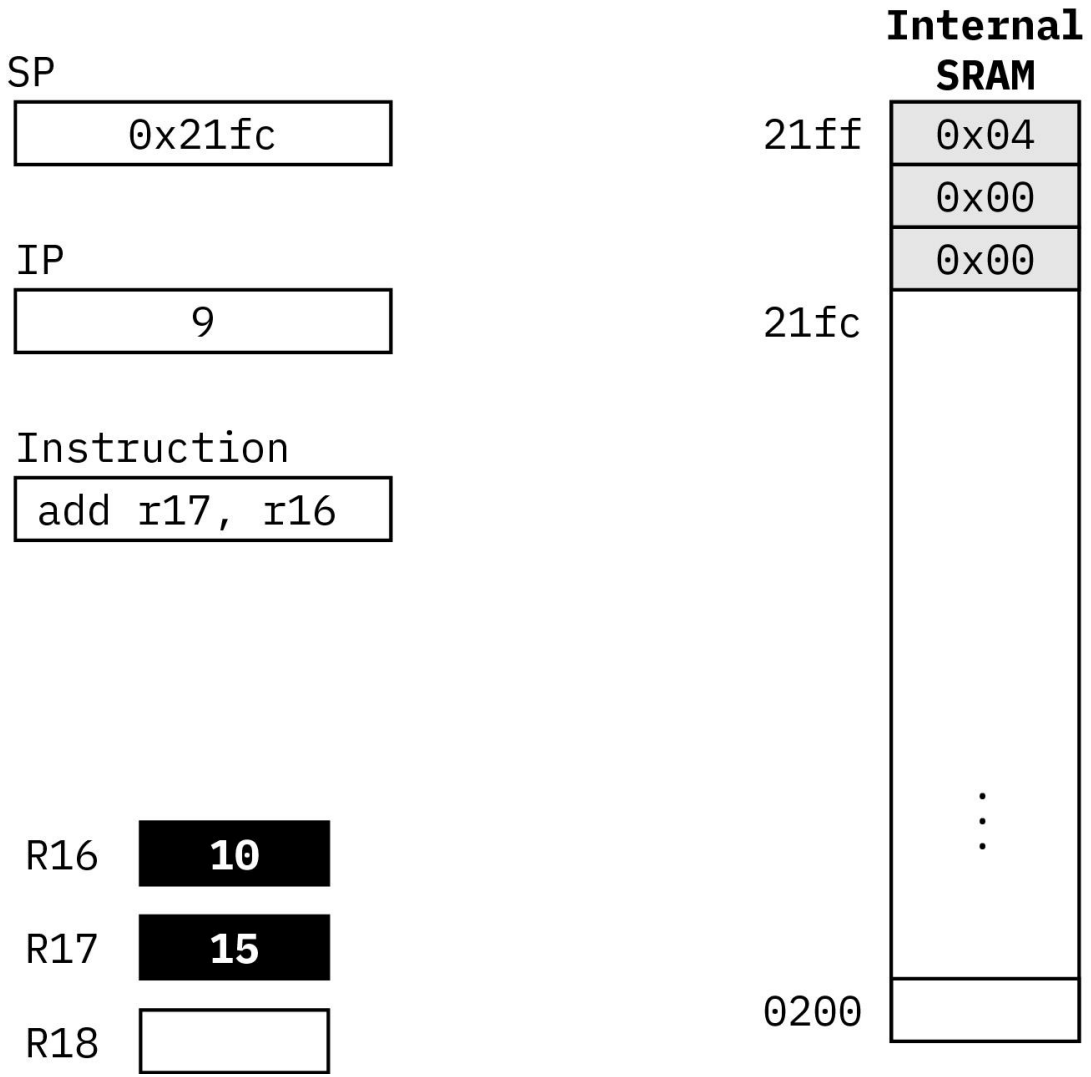
Start cycle

```
.org 0
    ldi r16, 10
    ldi r17, 5
    call foo
    ldi r18, 1
    add r17, r18
    call foo

stop:
    rjmp stop

foo:
    add r17, r16
    ret
```



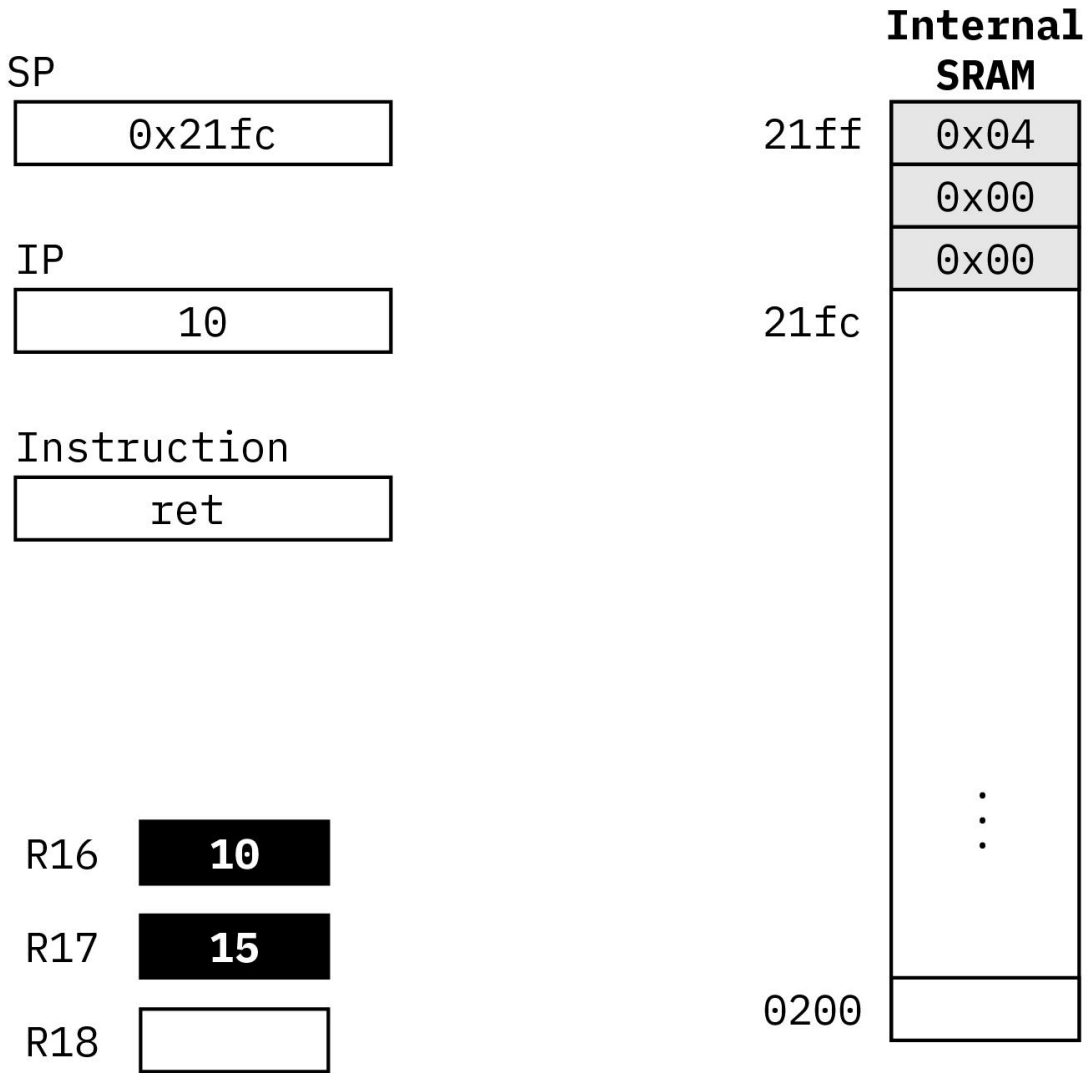
Finish cycle

```
.org 0
    ldi r16, 10
    ldi r17, 5
    call foo
    ldi r18, 1
    add r17, r18
    call foo

stop:
    rjmp stop

foo:
    add r17, r16
    ret
```



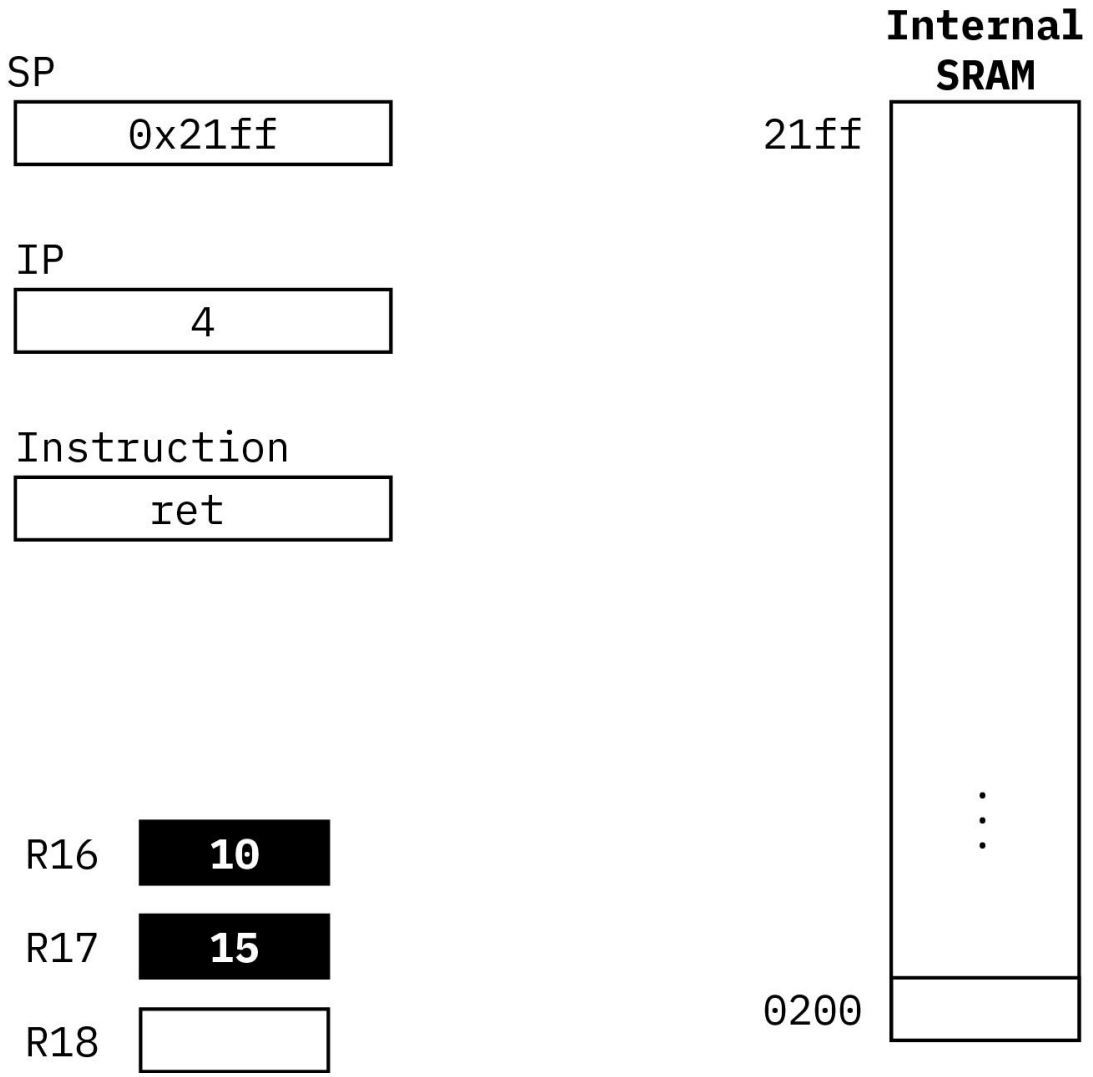
Start cycle

```
.org 0
    ldi r16, 10
    ldi r17, 5
    call foo
    ldi r18, 1
    add r17, r18
    call foo

stop:
    rjmp stop

foo:
    add r17, r16
    ret
```



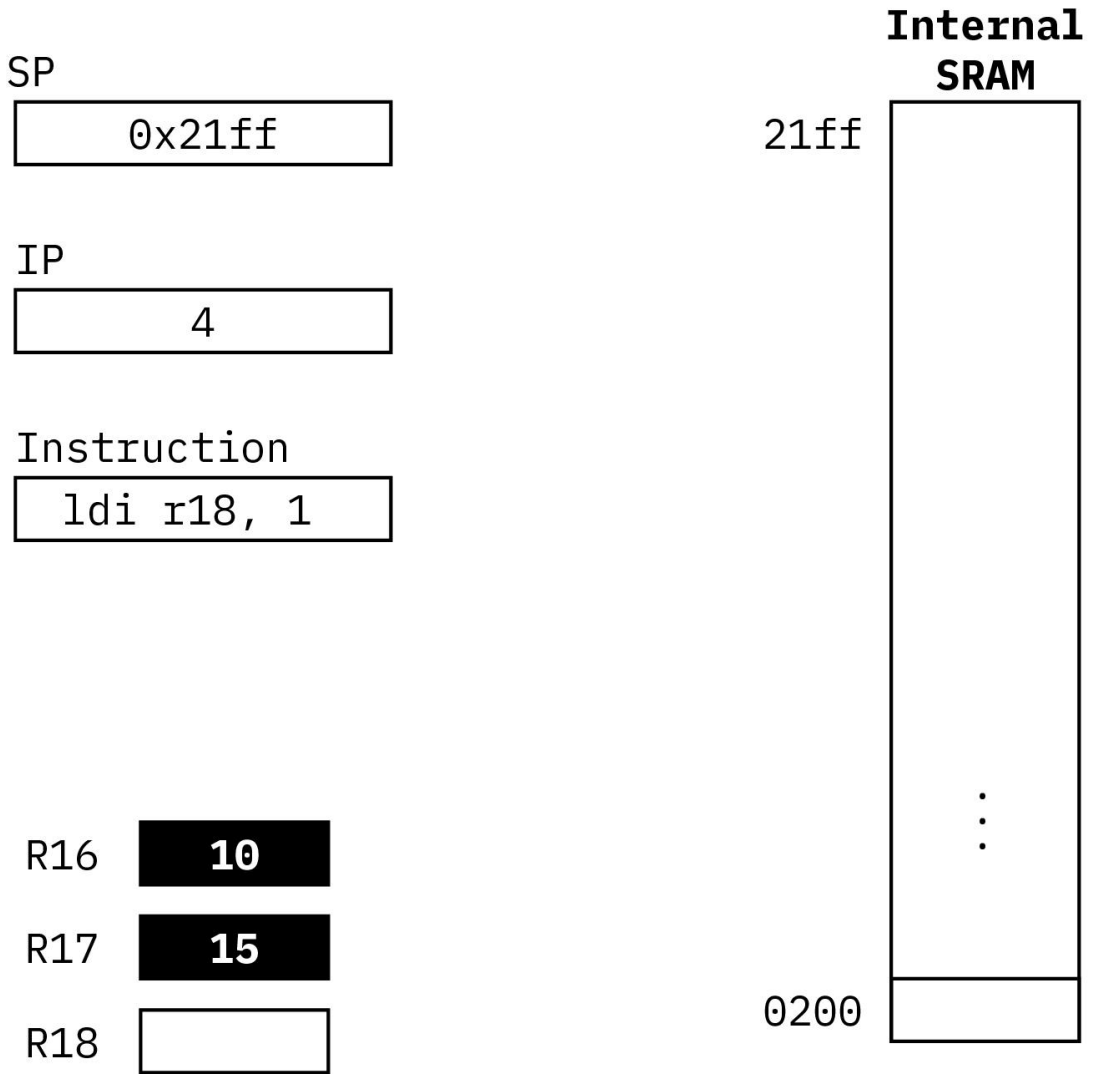
Finish cycle

```
.org 0
    ldi r16, 10
    ldi r17, 5
    call foo
    ldi r18, 1
    add r17, r18
    call foo

stop:
    rjmp stop

foo:
    add r17, r16
    ret
```



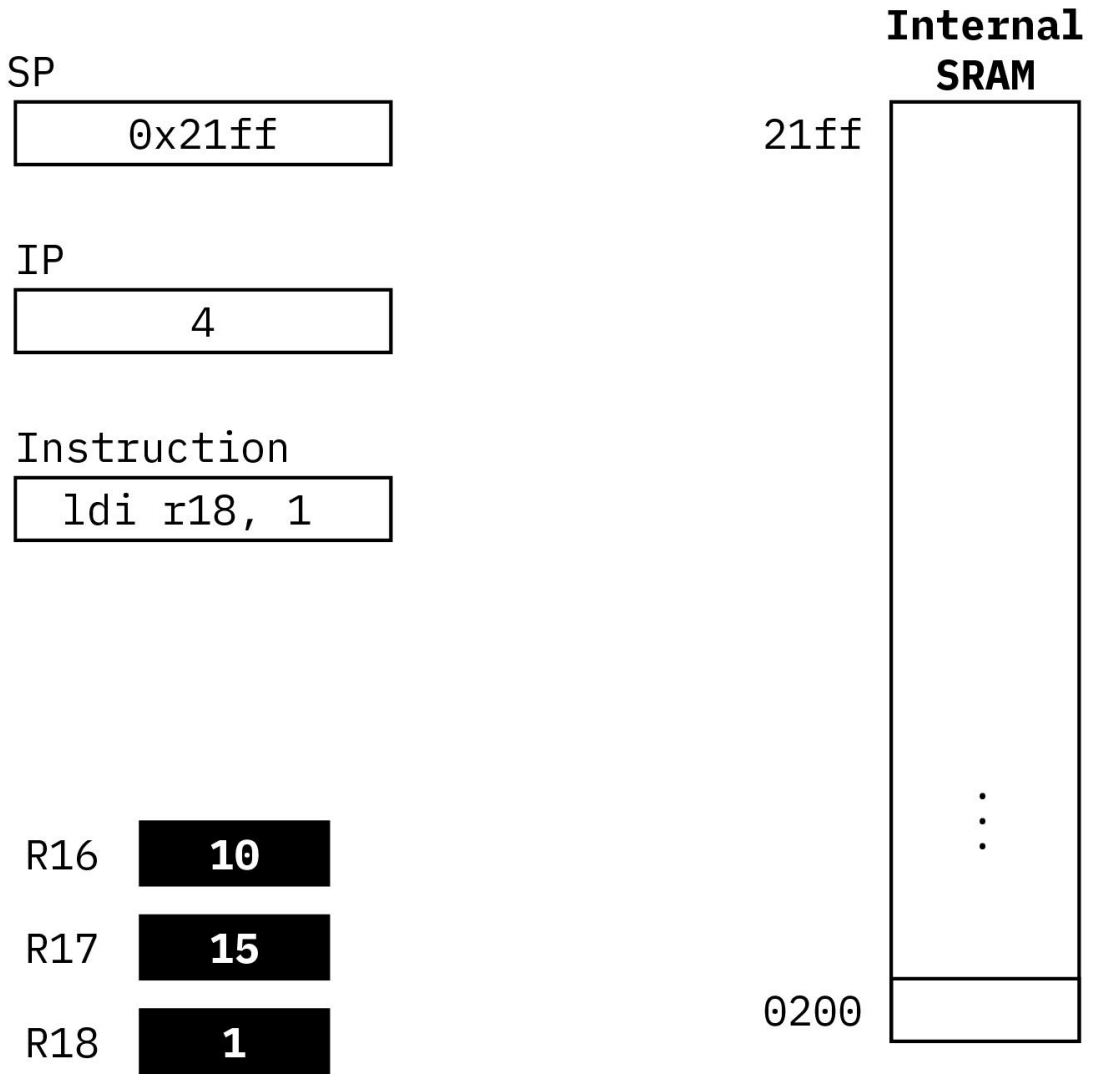
Start cycle

```
.org 0
    ldi r16, 10
    ldi r17, 5
    call foo
    ldi r18, 1
    add r17, r18
    call foo

stop:
    rjmp stop

foo:
    add r17, r16
    ret
```



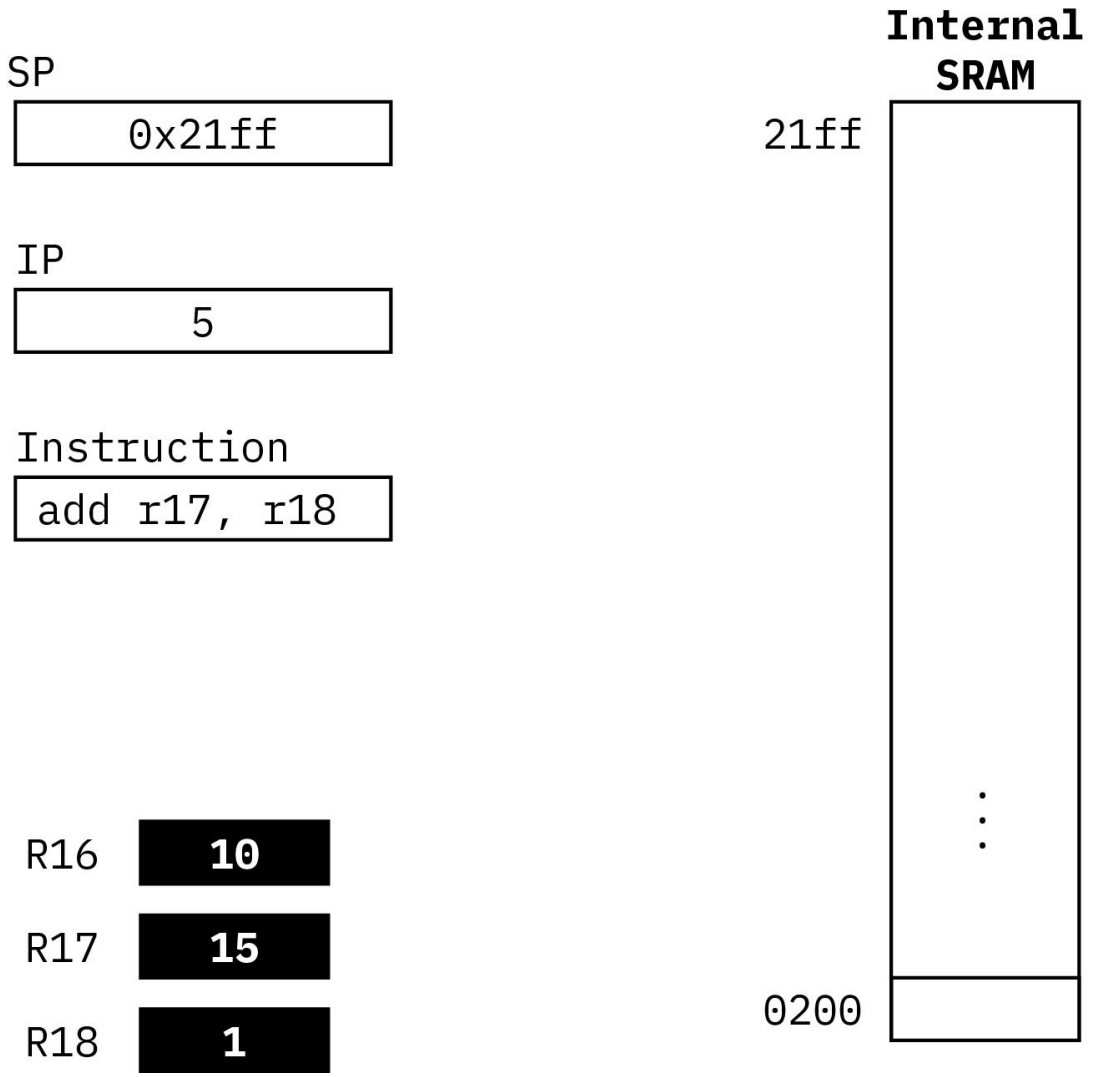
Finish cycle

```
.org 0
    ldi r16, 10
    ldi r17, 5
    call foo
    ldi r18, 1
    add r17, r18
    call foo

stop:
    rjmp stop

foo:
    add r17, r16
    ret
```



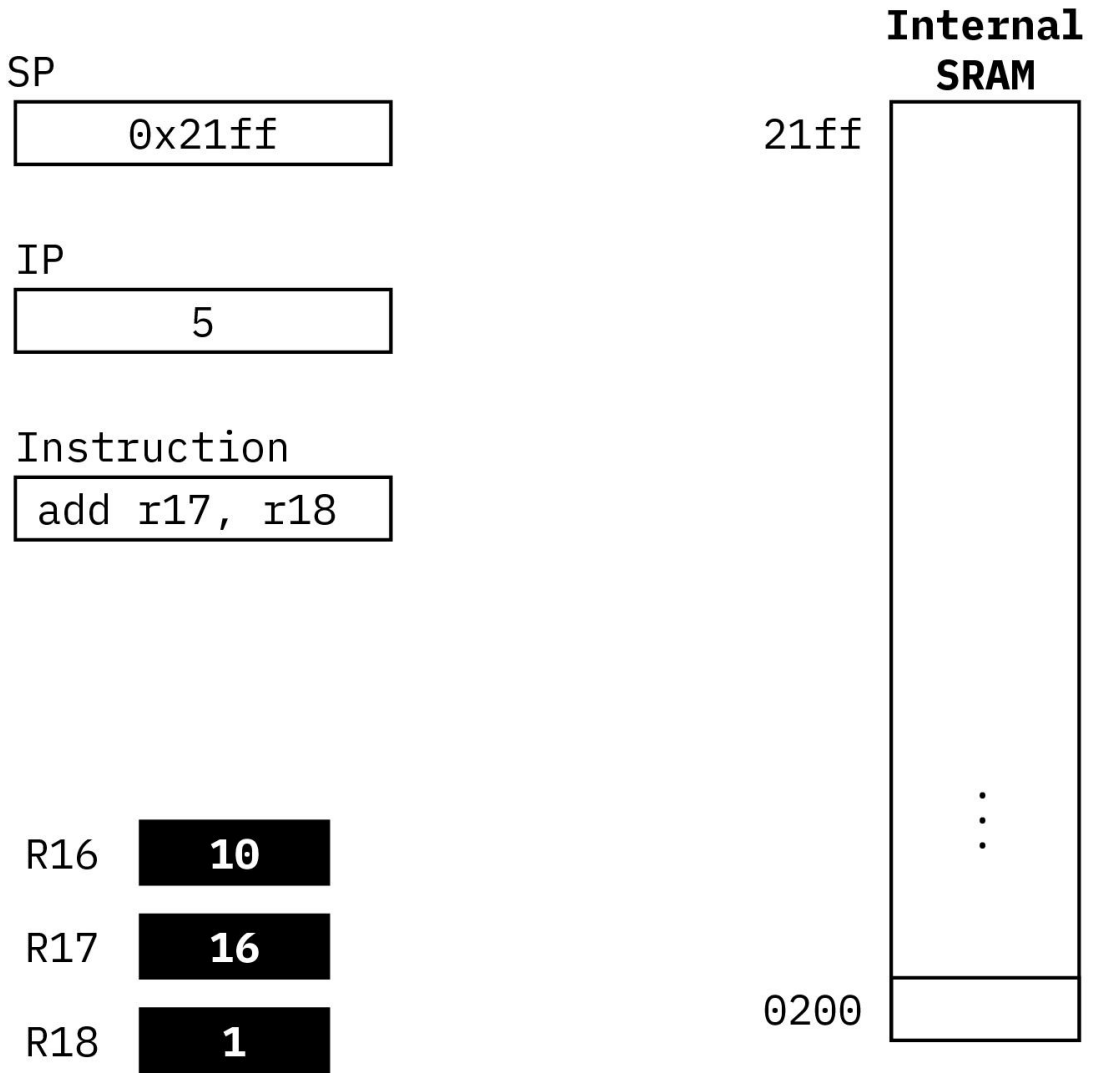
Start cycle

```
.org 0
    ldi r16, 10
    ldi r17, 5
    call foo
    ldi r18, 1
    add r17, r18
    call foo

stop:
    rjmp stop

foo:
    add r17, r16
    ret
```



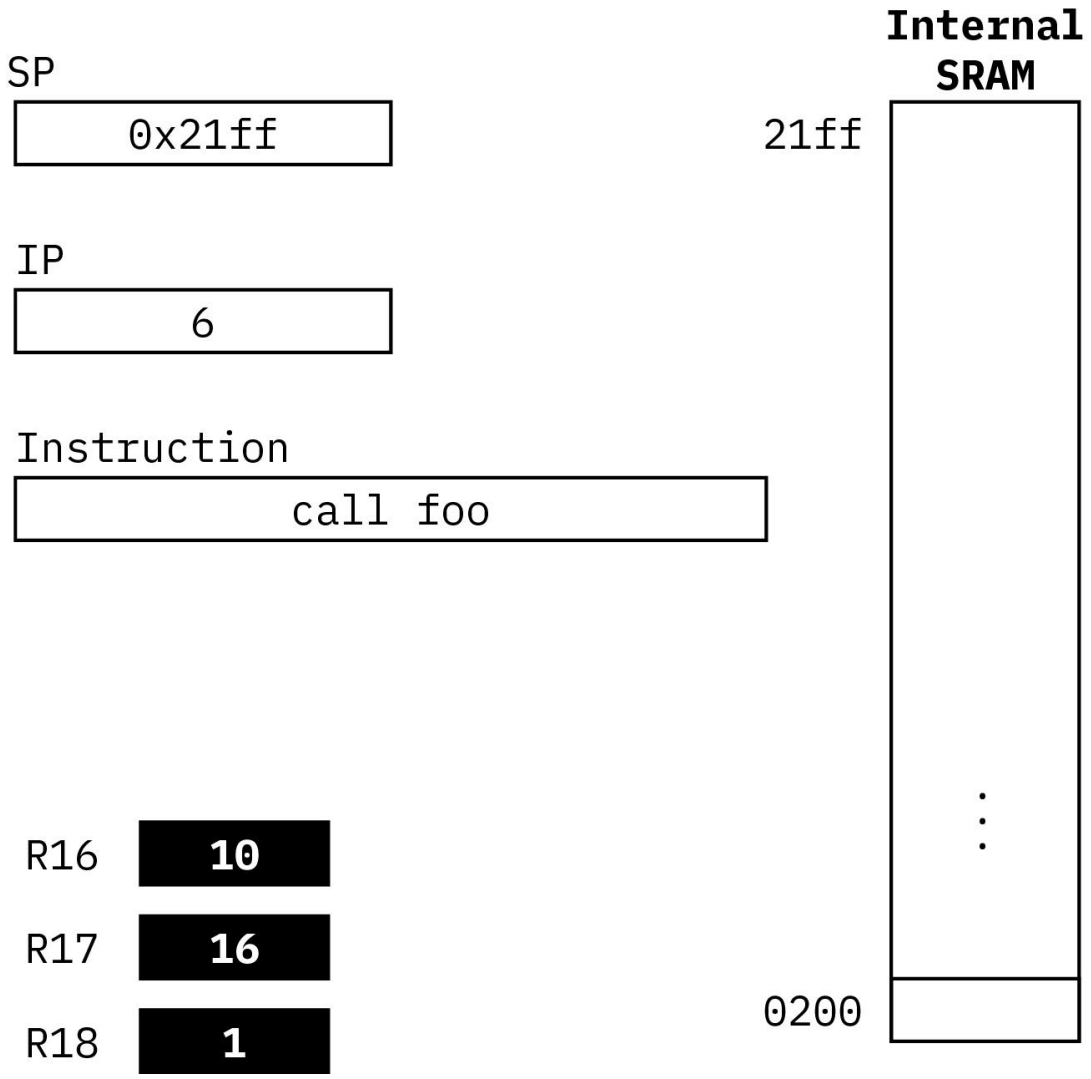
Finish cycle

```
.org 0
    ldi r16, 10
    ldi r17, 5
    call foo
    ldi r18, 1
    add r17, r18
    call foo

stop:
    rjmp stop

foo:
    add r17, r16
    ret
```



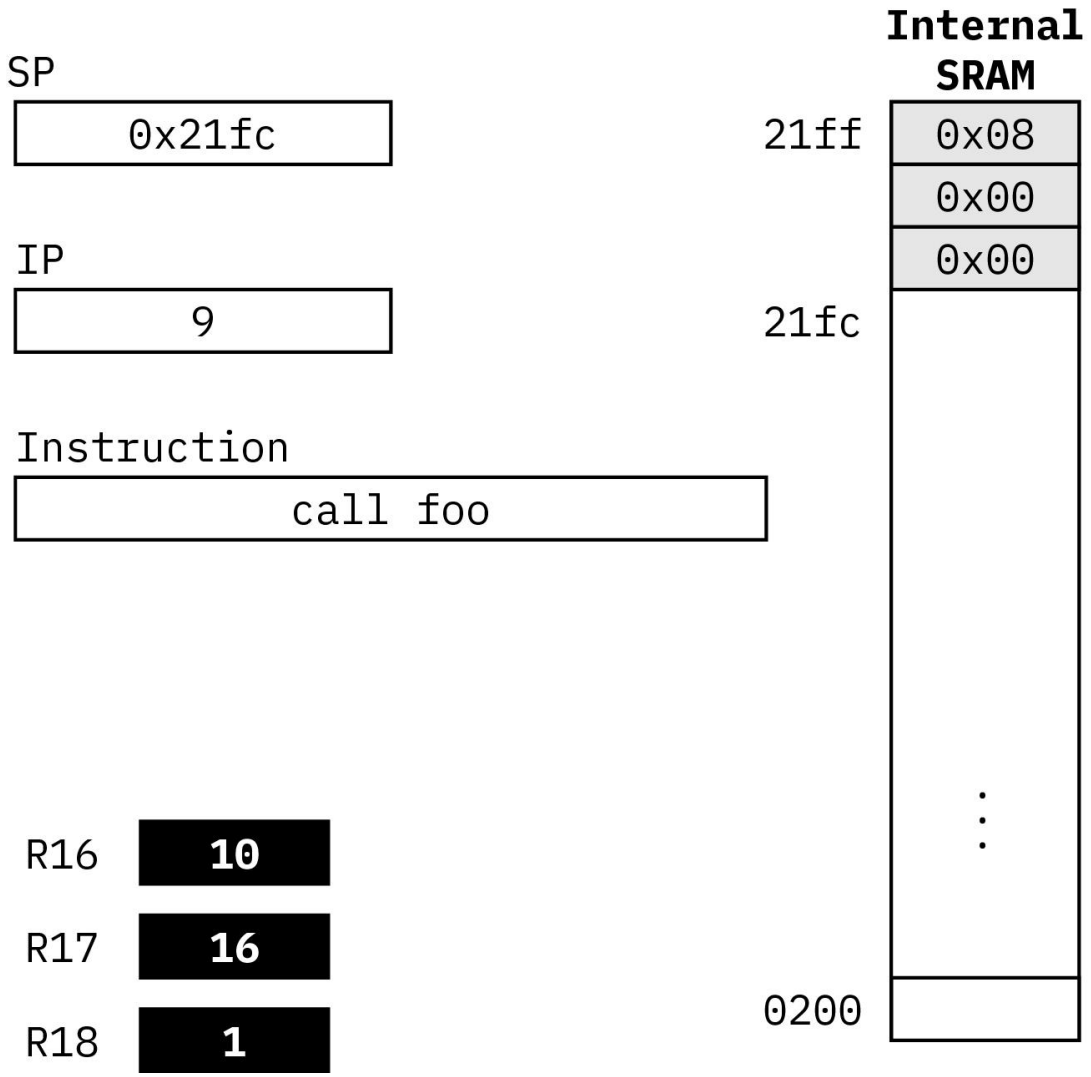
Start cycle

```
.org 0
    ldi r16, 10
    ldi r17, 5
    call foo
    ldi r18, 1
    add r17, r18
    call foo

stop:
    rjmp stop

foo:
    add r17, r16
    ret
```



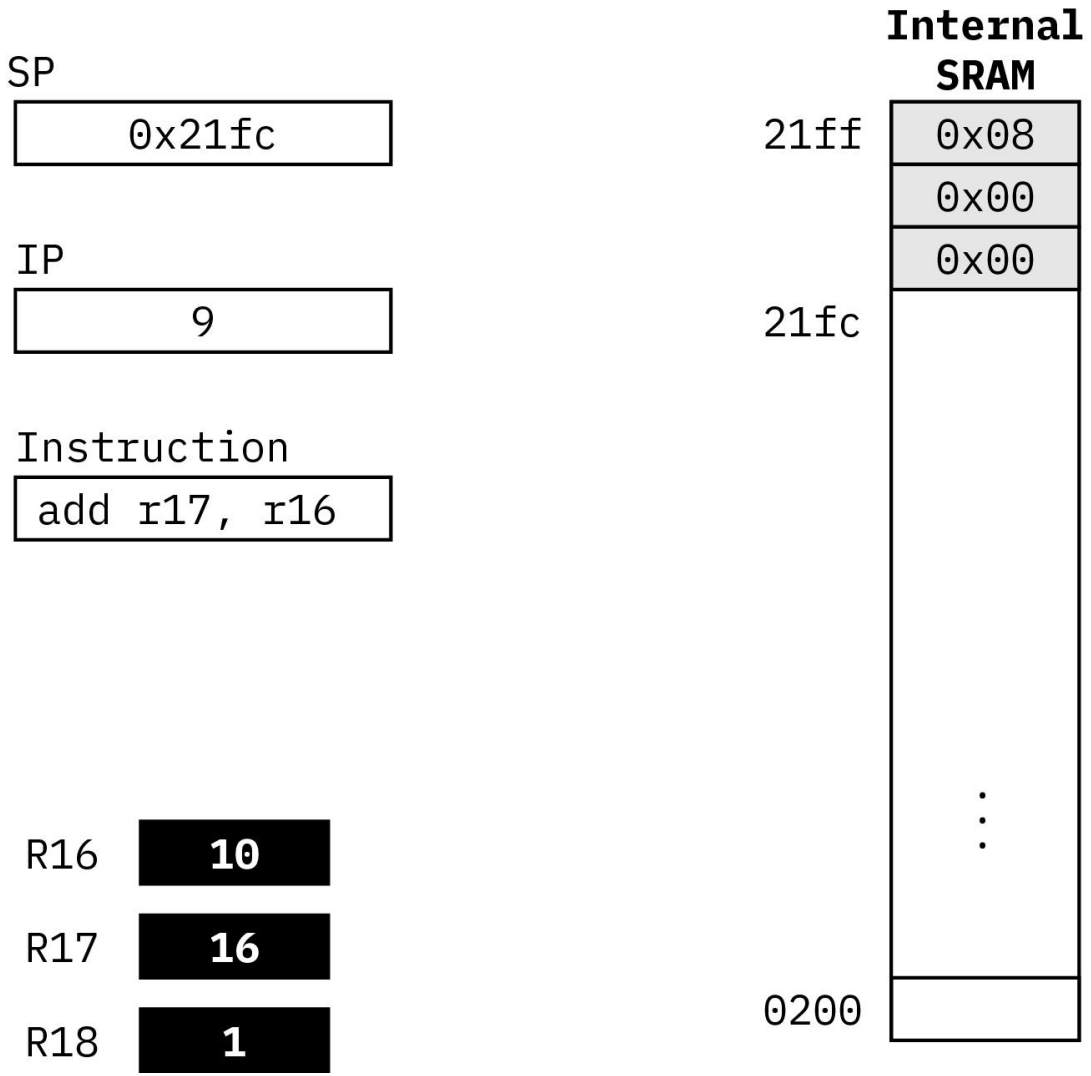
Finish cycle

```
.org 0
    ldi r16, 10
    ldi r17, 5
    call foo
    ldi r18, 1
    add r17, r18
    call foo

stop:
    rjmp stop

foo:
    add r17, r16
    ret
```



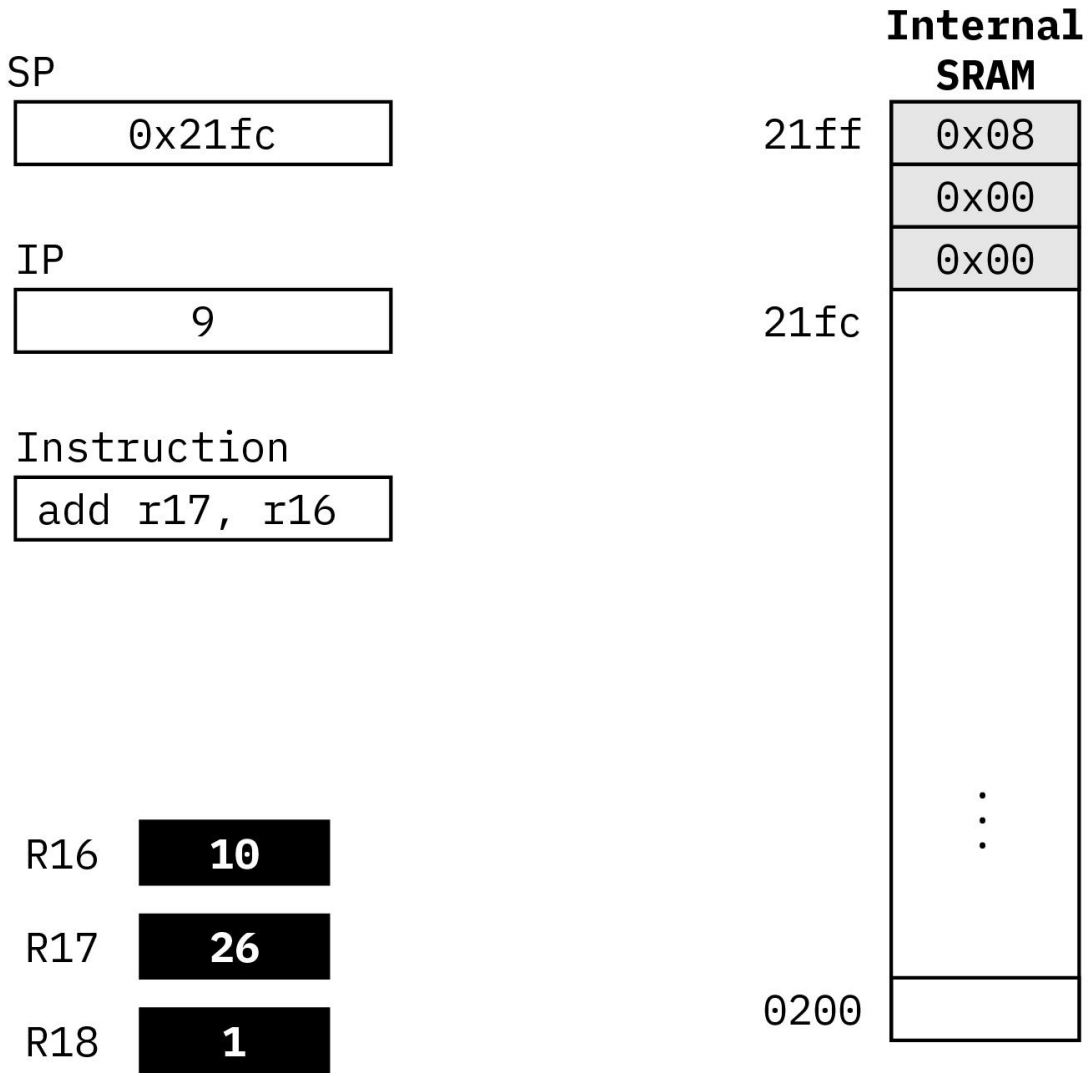
Start cycle

```
.org 0
    ldi r16, 10
    ldi r17, 5
    call foo
    ldi r18, 1
    add r17, r18
    call foo

stop:
    rjmp stop

foo:
    add r17, r16
    ret
```



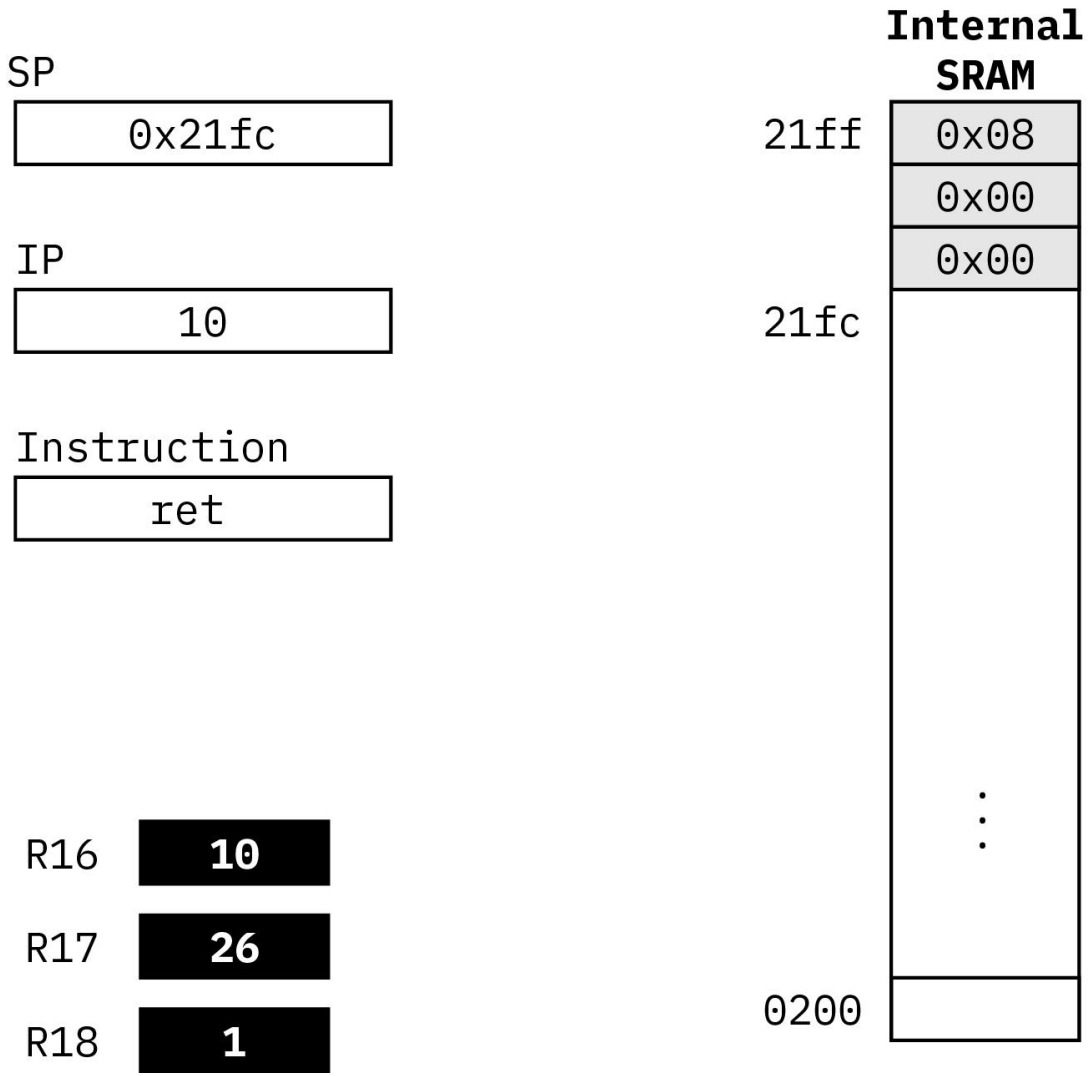
Finish cycle

```
.org 0
    ldi r16, 10
    ldi r17, 5
    call foo
    ldi r18, 1
    add r17, r18
    call foo

stop:
    rjmp stop

foo:
    add r17, r16
    ret
```



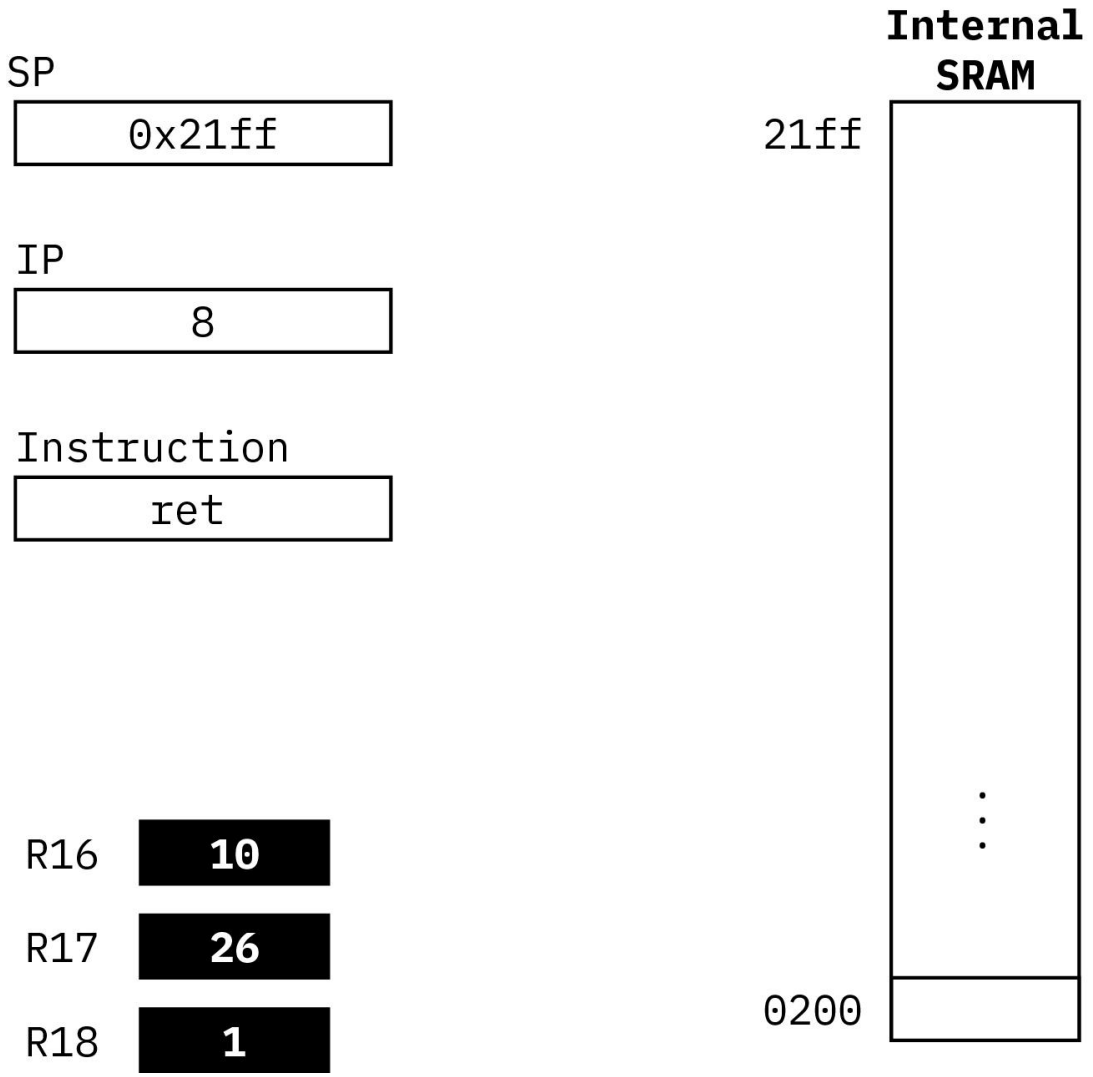
Start cycle

```
.org 0
    ldi r16, 10
    ldi r17, 5
    call foo
    ldi r18, 1
    add r17, r18
    call foo

stop:
    rjmp stop

foo:
    add r17, r16
    ret
```



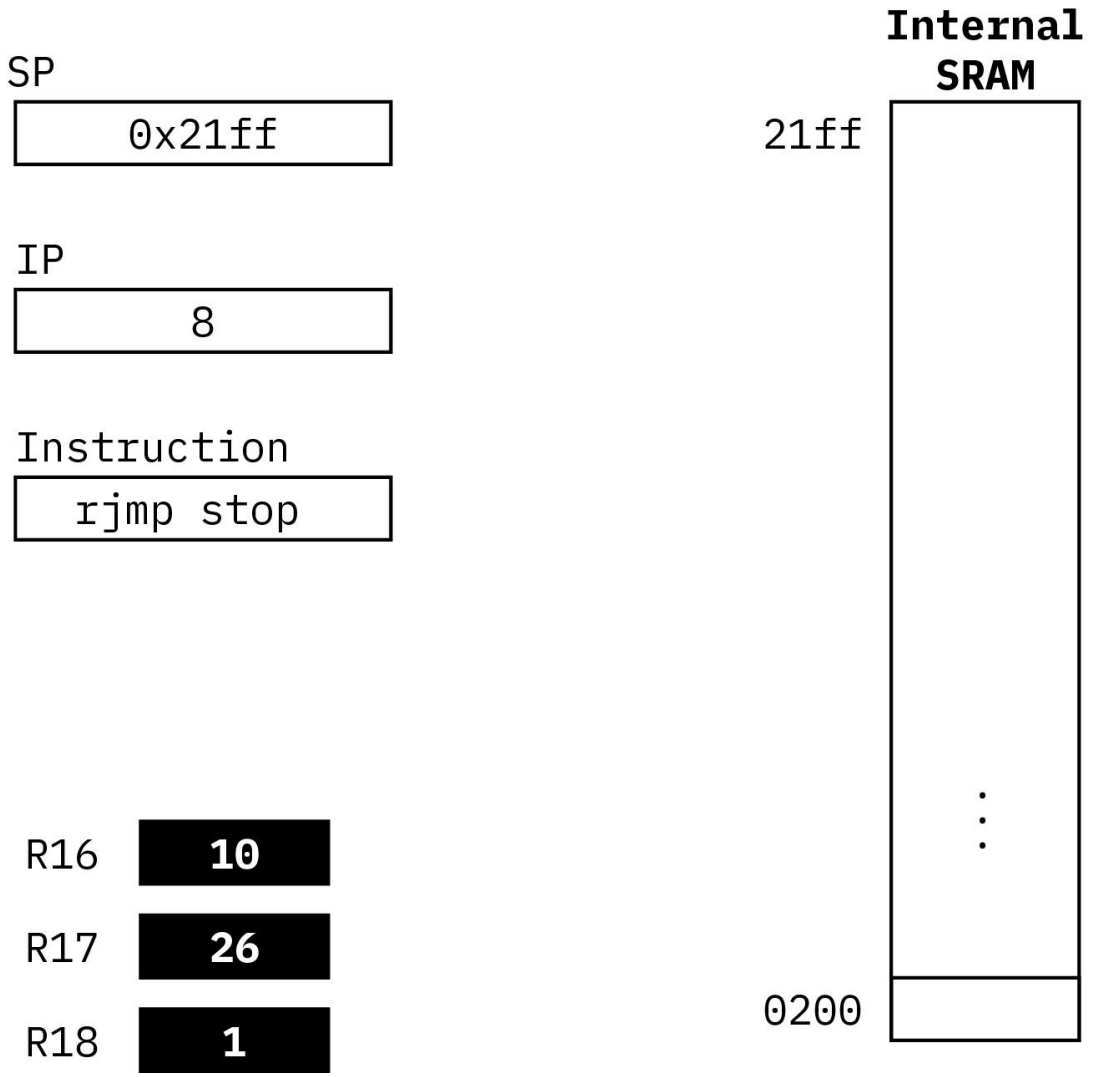
Finish cycle

```
.org 0
    ldi r16, 10
    ldi r17, 5
    call foo
    ldi r18, 1
    add r17, r18
    call foo

stop:
    rjmp stop

foo:
    add r17, r16
    ret
```



Start cycle

```
.org 0
    ldi r16, 10
    ldi r17, 5
    call foo
    ldi r18, 1
    add r17, r18
    call foo

stop:
    rjmp stop

foo:
    add r17, r16
    ret
```



Any Questions?