CSC 230 - Summer 2018

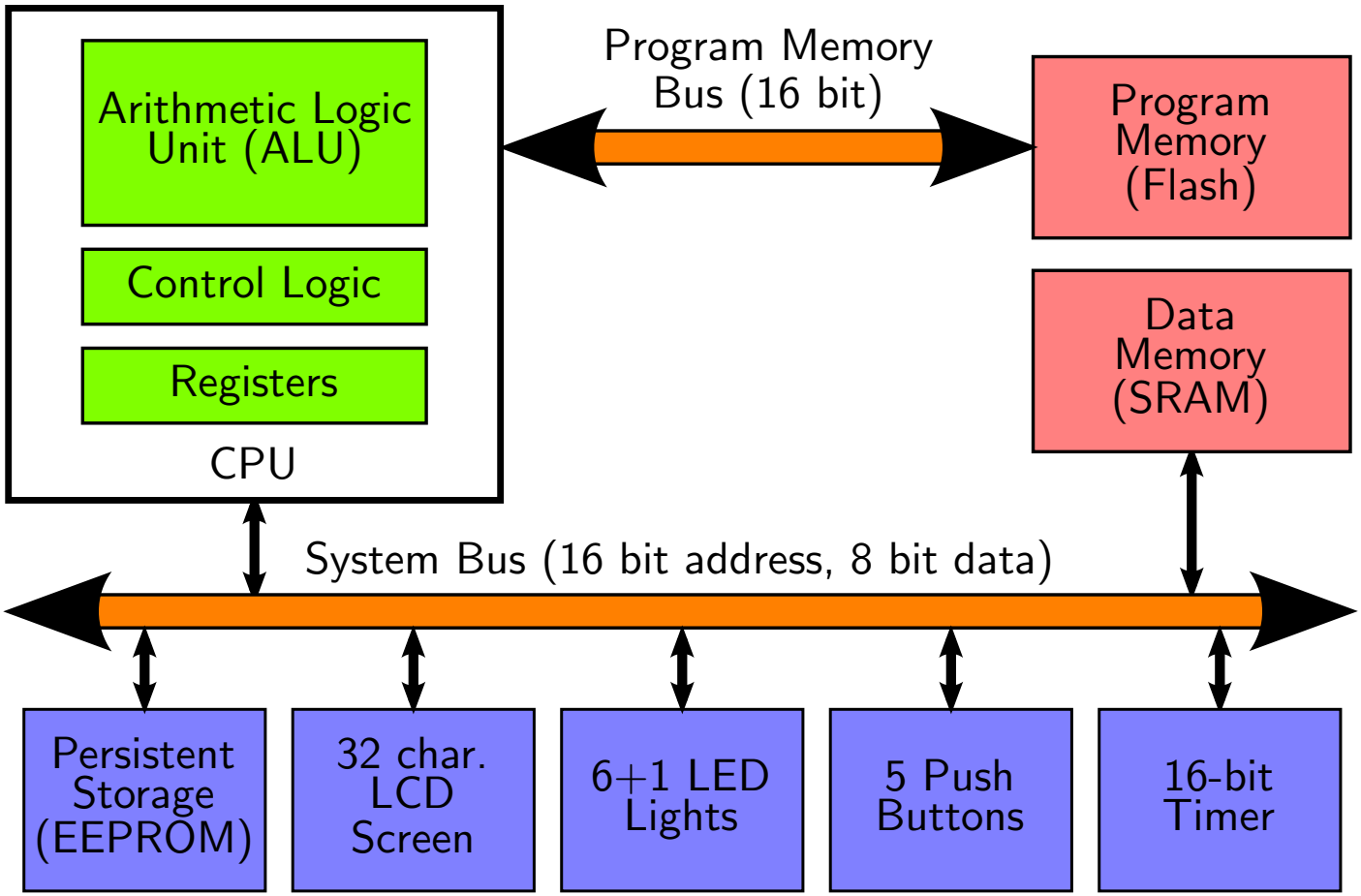
Memory

Bill Bird

Department of Computer Science
University of Victoria

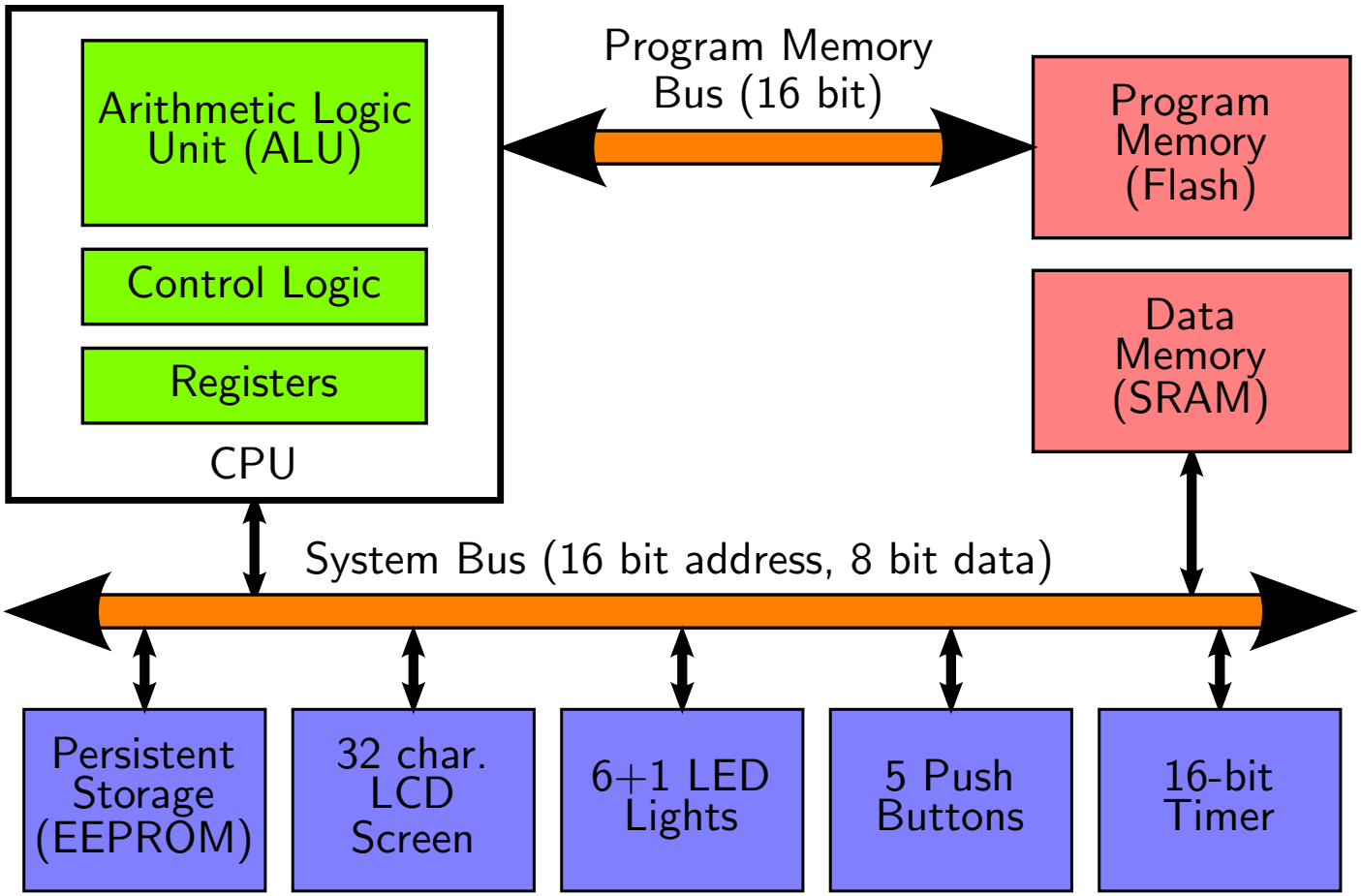
June 7, 2018

AVR Memory Buses (1)



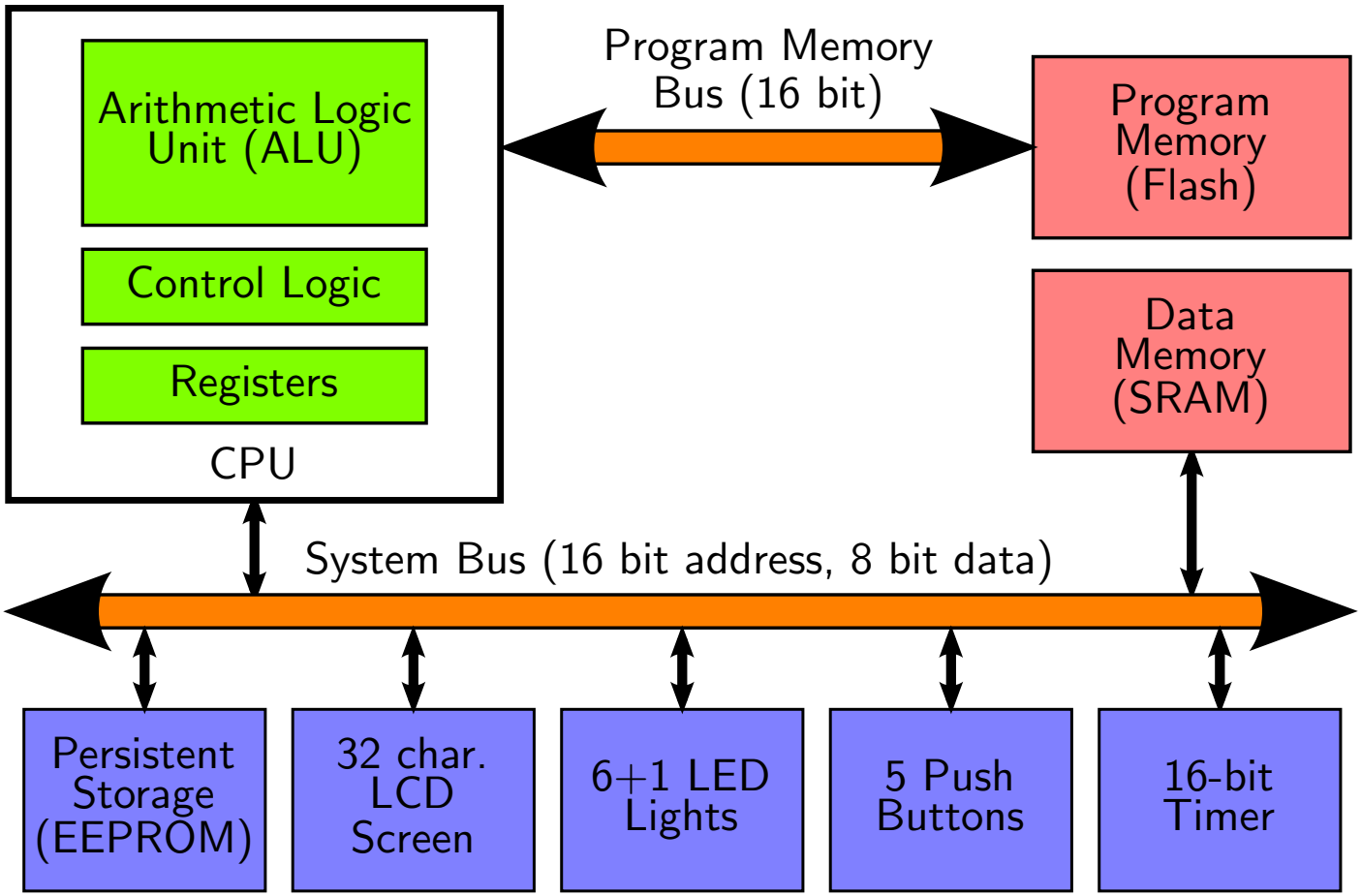
Recall that AVR is a **Harvard architecture** with separate memory spaces for code (**program memory**) and data (**data memory**).

AVR Memory Buses (2)



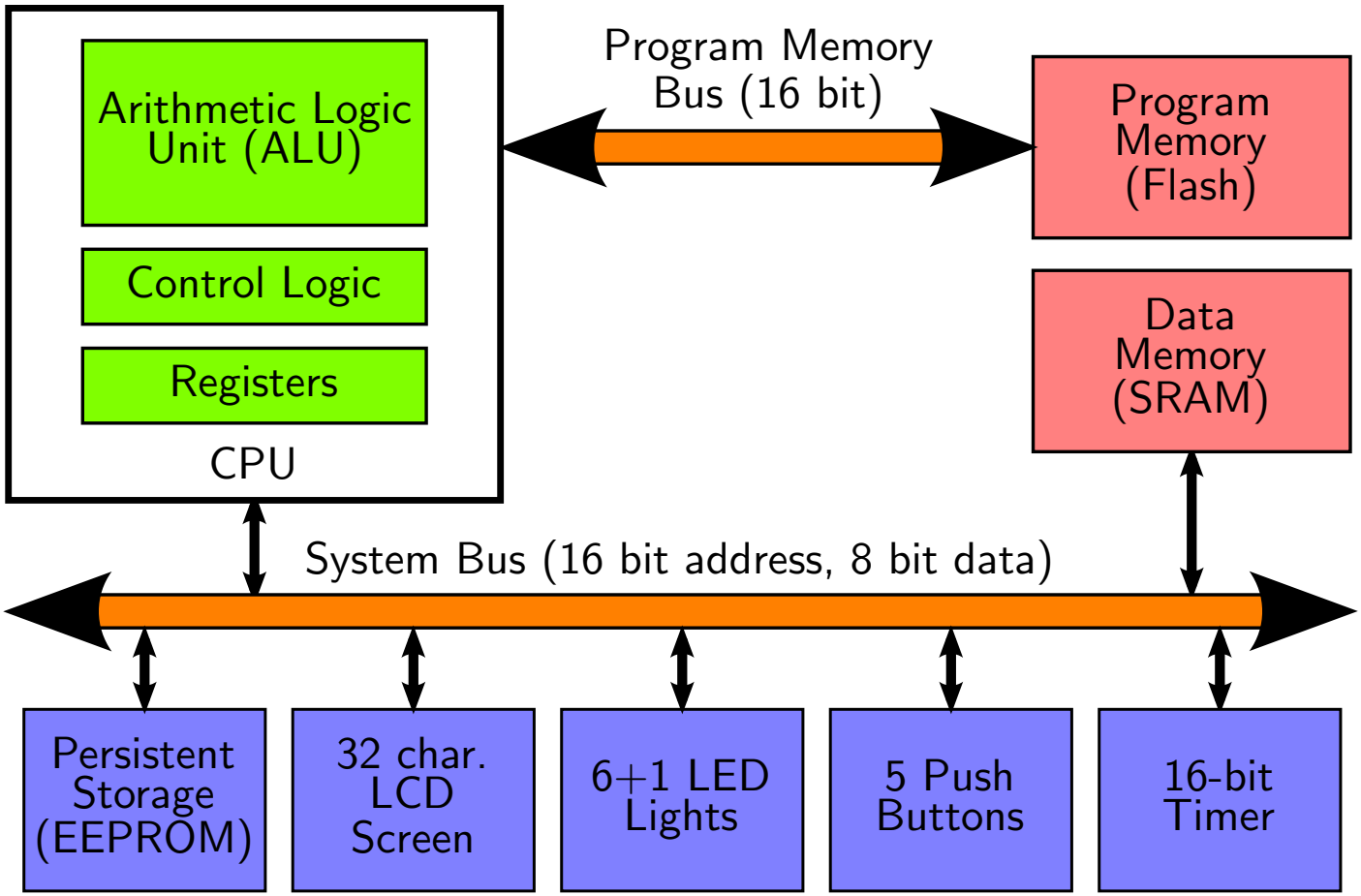
The processor communicates with each memory space using one of the two buses shown above. Notice that data memory shares a bus with all of the peripheral devices.

AVR Memory Buses (3)



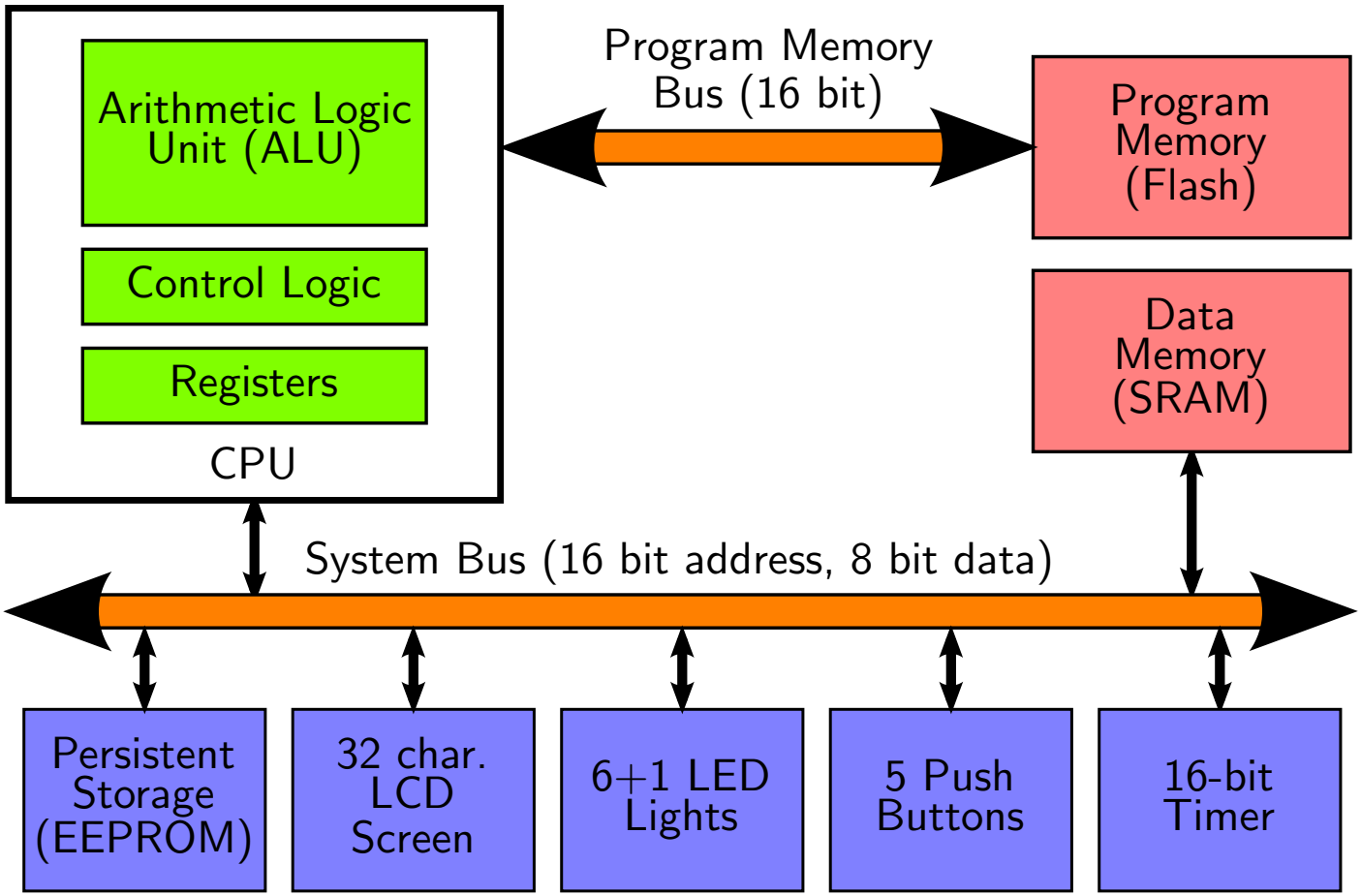
Each of the two memory buses can be accessed by a different set of instructions. You may notice that memory can **only** be accessed by dedicated memory-access instructions.

AVR Memory Buses (4)



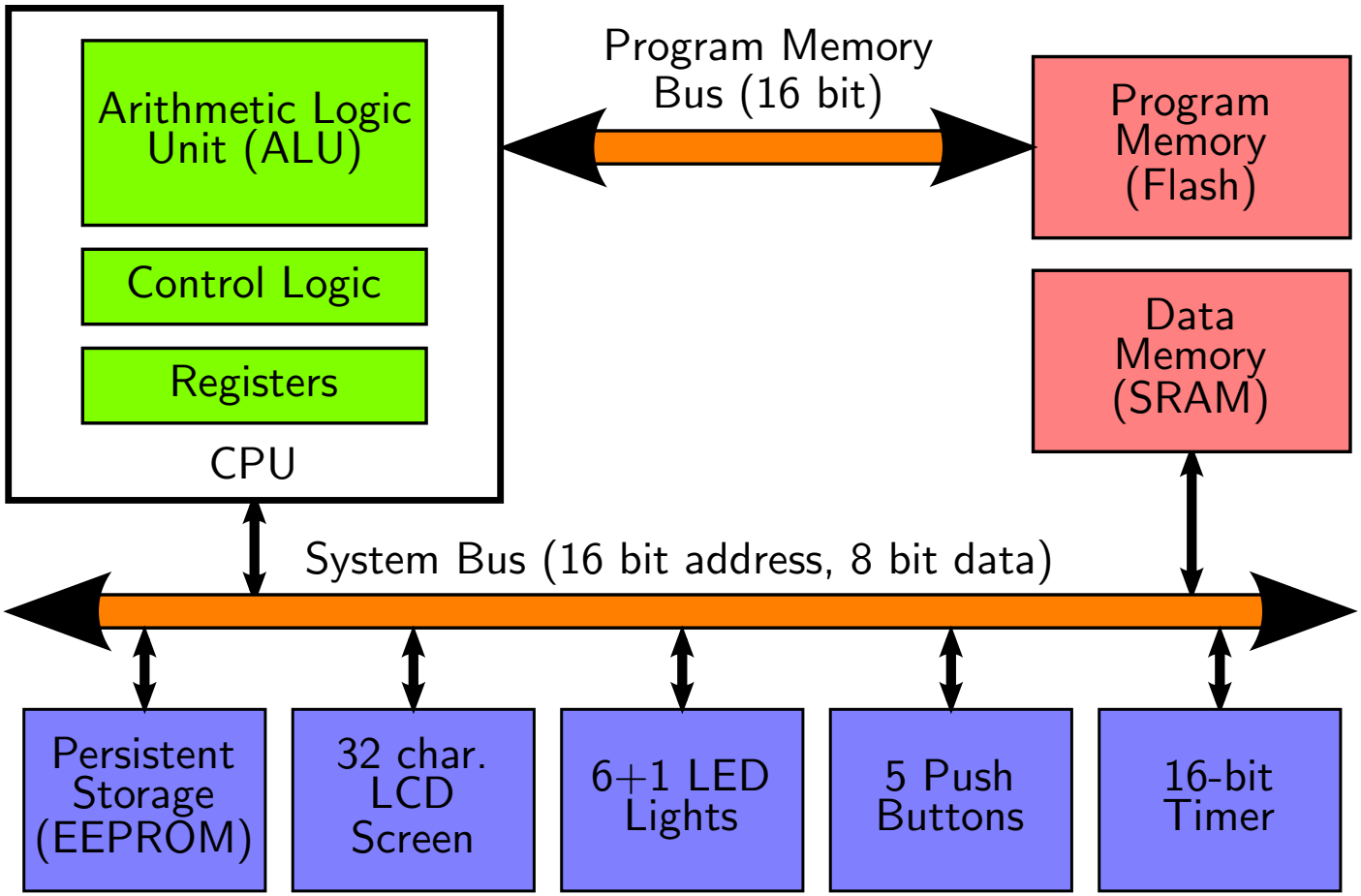
The `LDS`, `LD`, `STS` and `ST` instructions are used to load and store data with the main system bus. Each memory address for the system bus is 16 bits wide, and each location is 8 bits wide.

AVR Memory Buses (5)



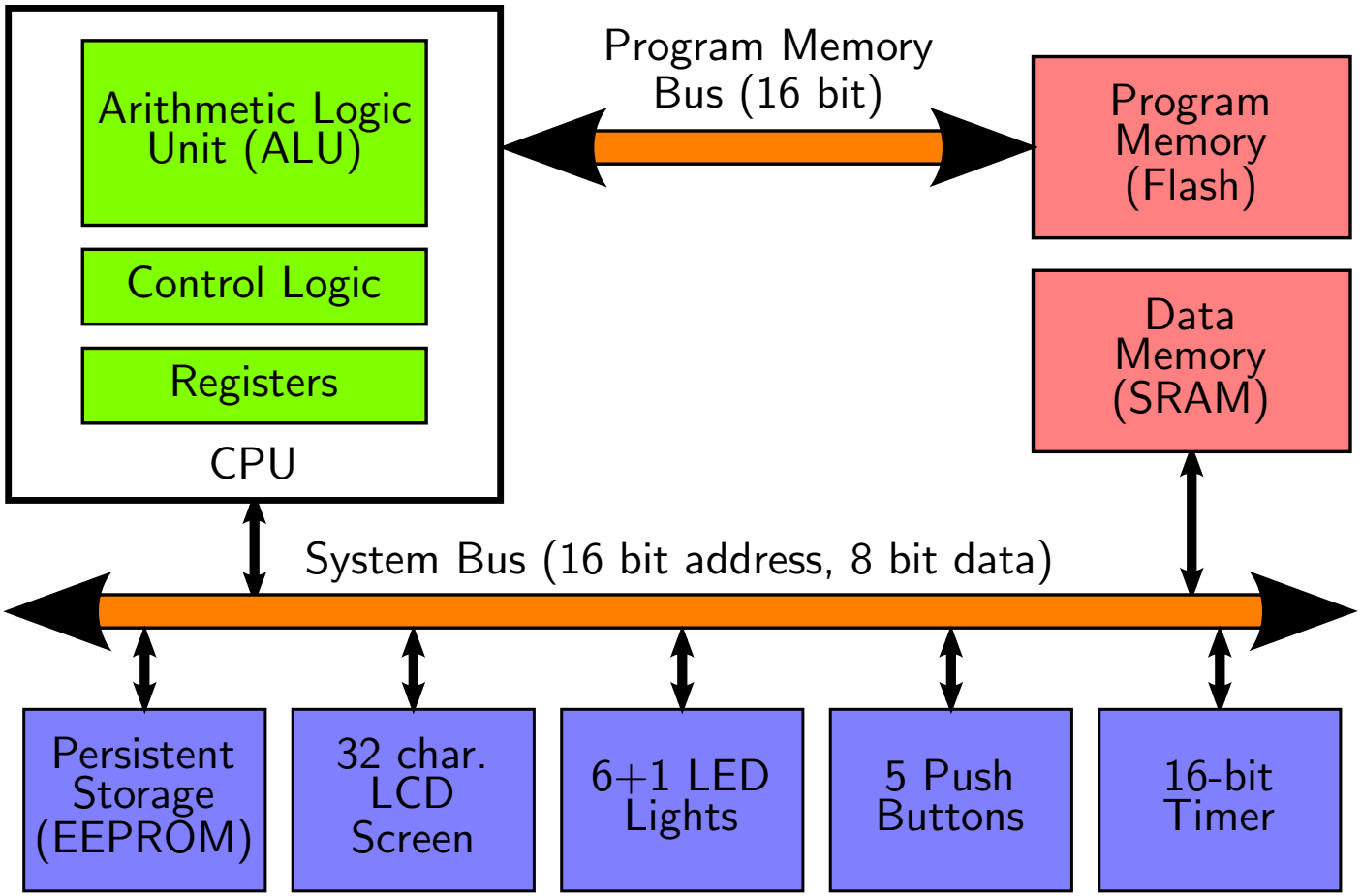
Question: If each memory address in data space is 16 bits wide, what is the maximum number of possible addresses that can be used?

AVR Memory Buses (6)



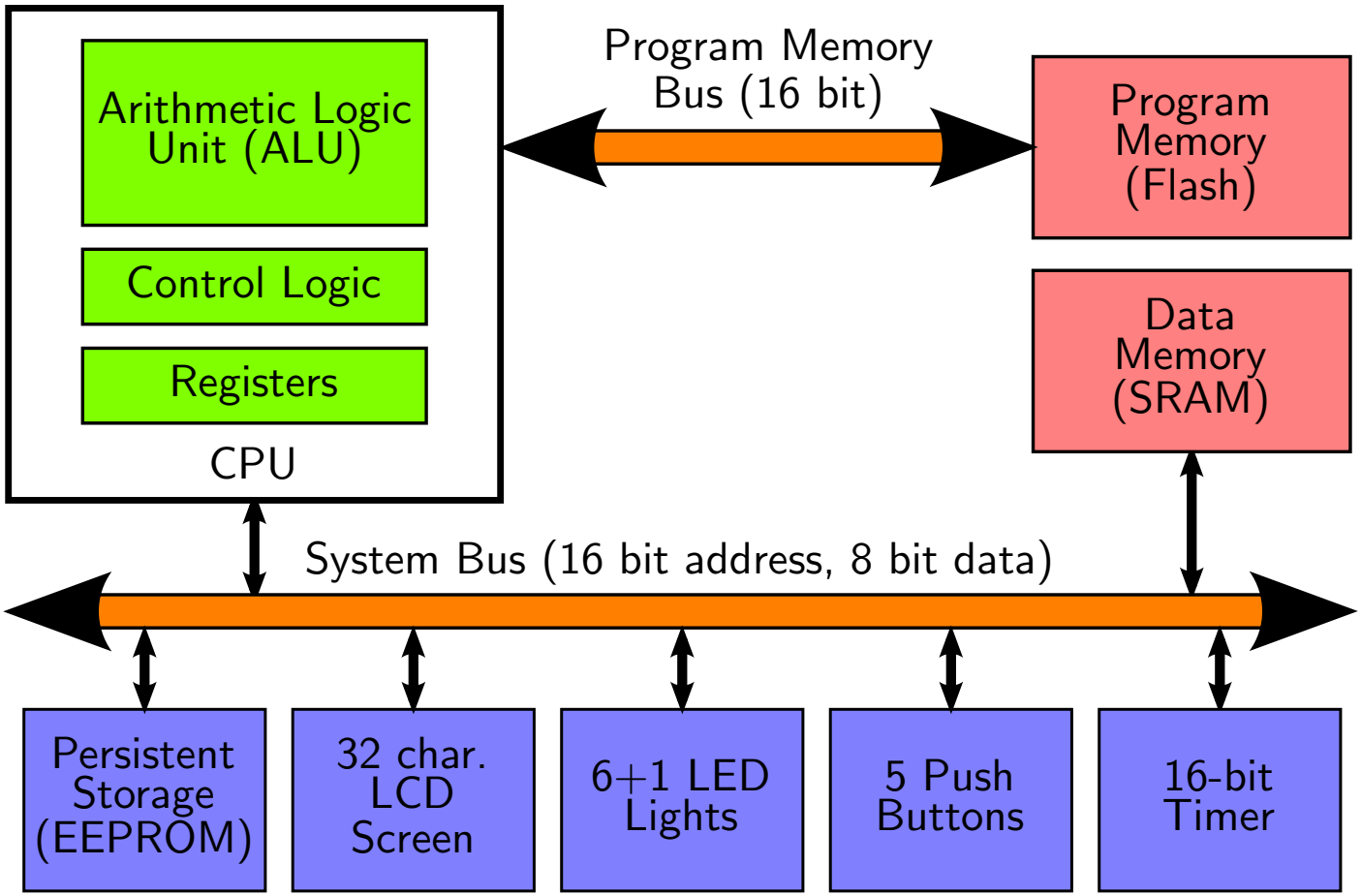
Since each memory location on the main bus is 8 bits wide, we can say that it is **byte addressable**. Note that memory addresses may be more than 16 bits wide on a byte addressable system.

AVR Memory Buses (7)



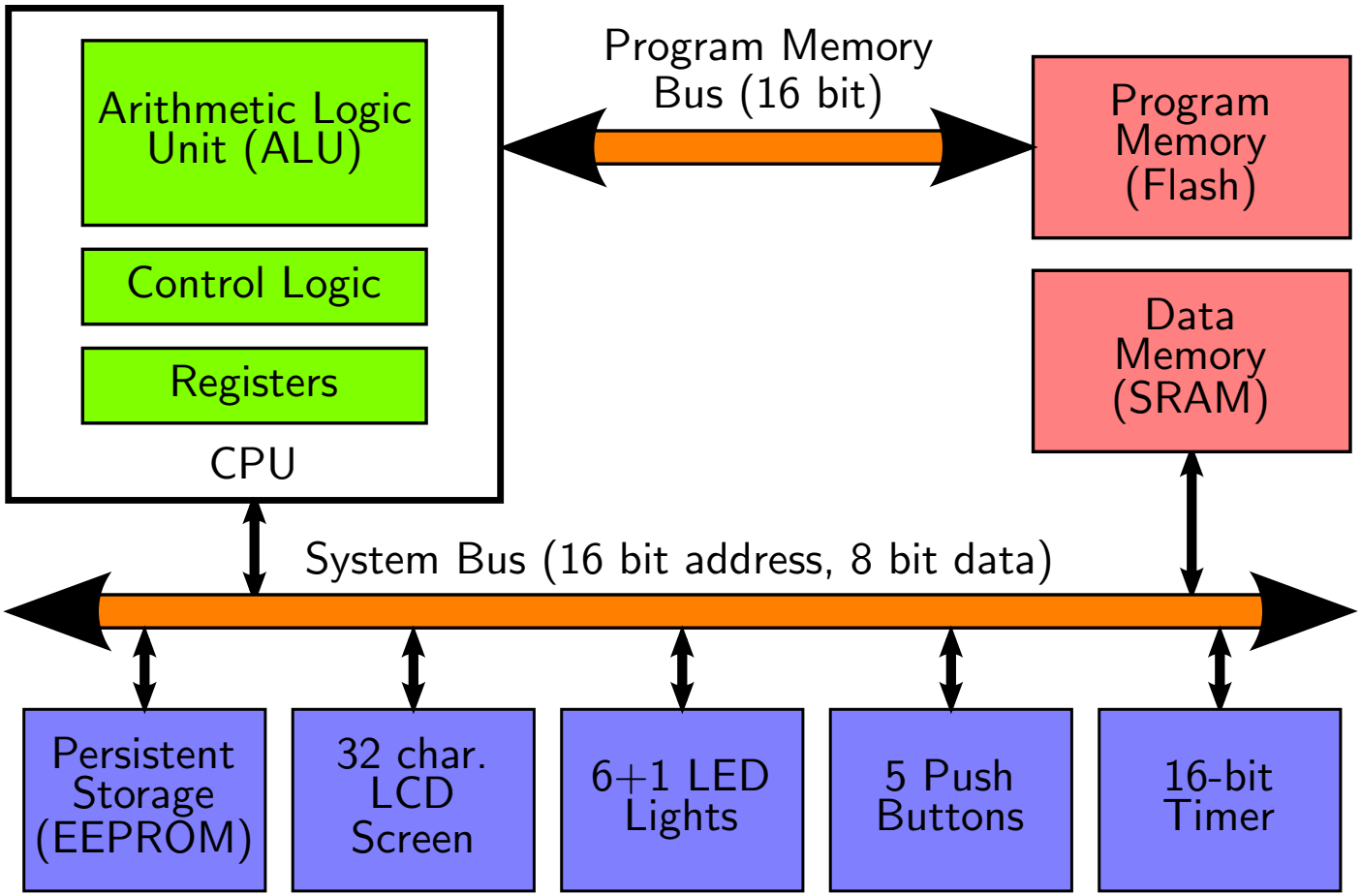
On the other hand, in program memory addresses are 17 bits wide, but each memory location is 16 bits wide (since most AVR instructions are 16 bits in size).

AVR Memory Buses (8)



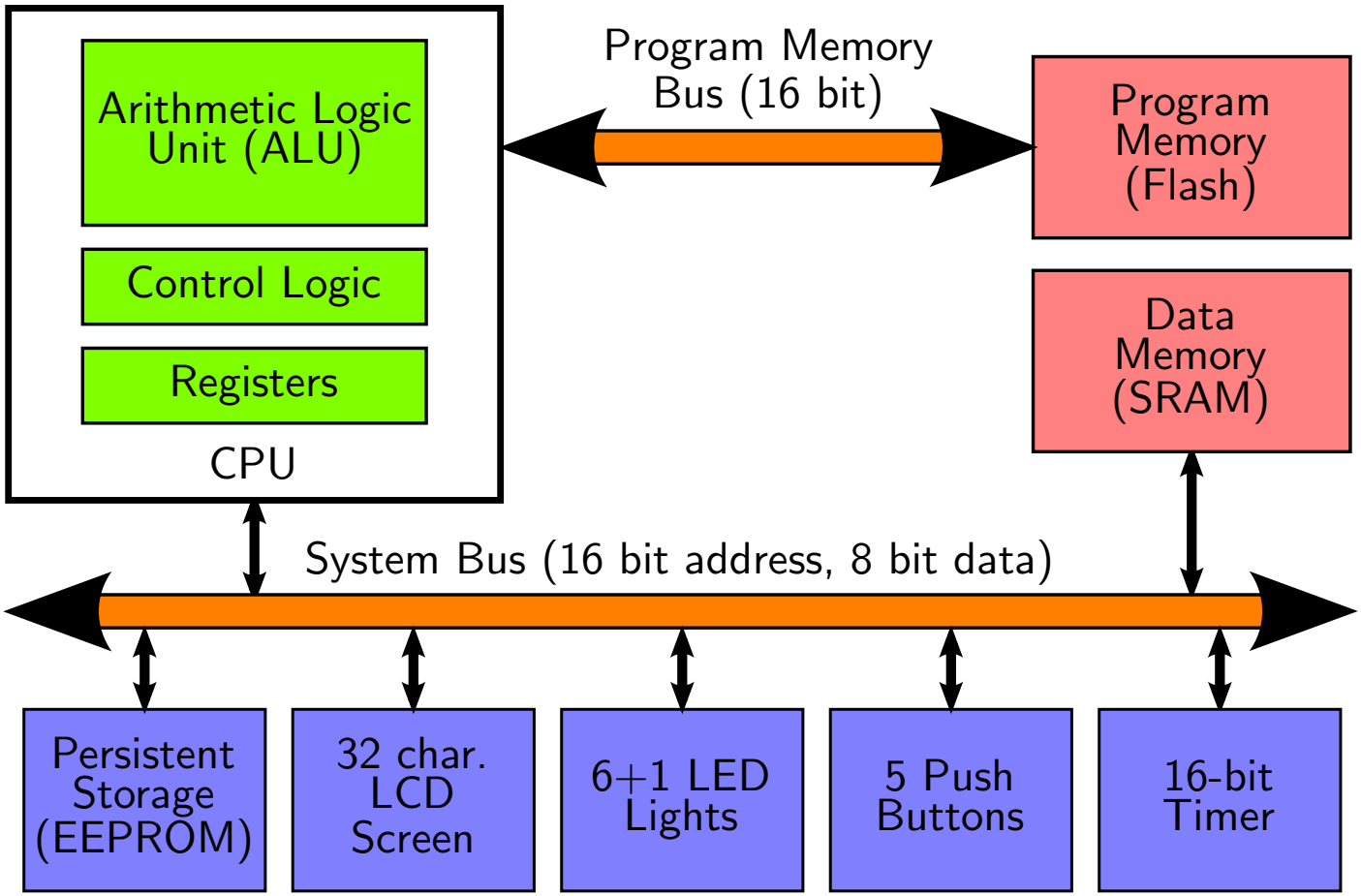
Question: What is the maximum size (in bytes) of program memory assuming 17 bit addresses and 16 bit locations?

AVR Memory Buses (9)



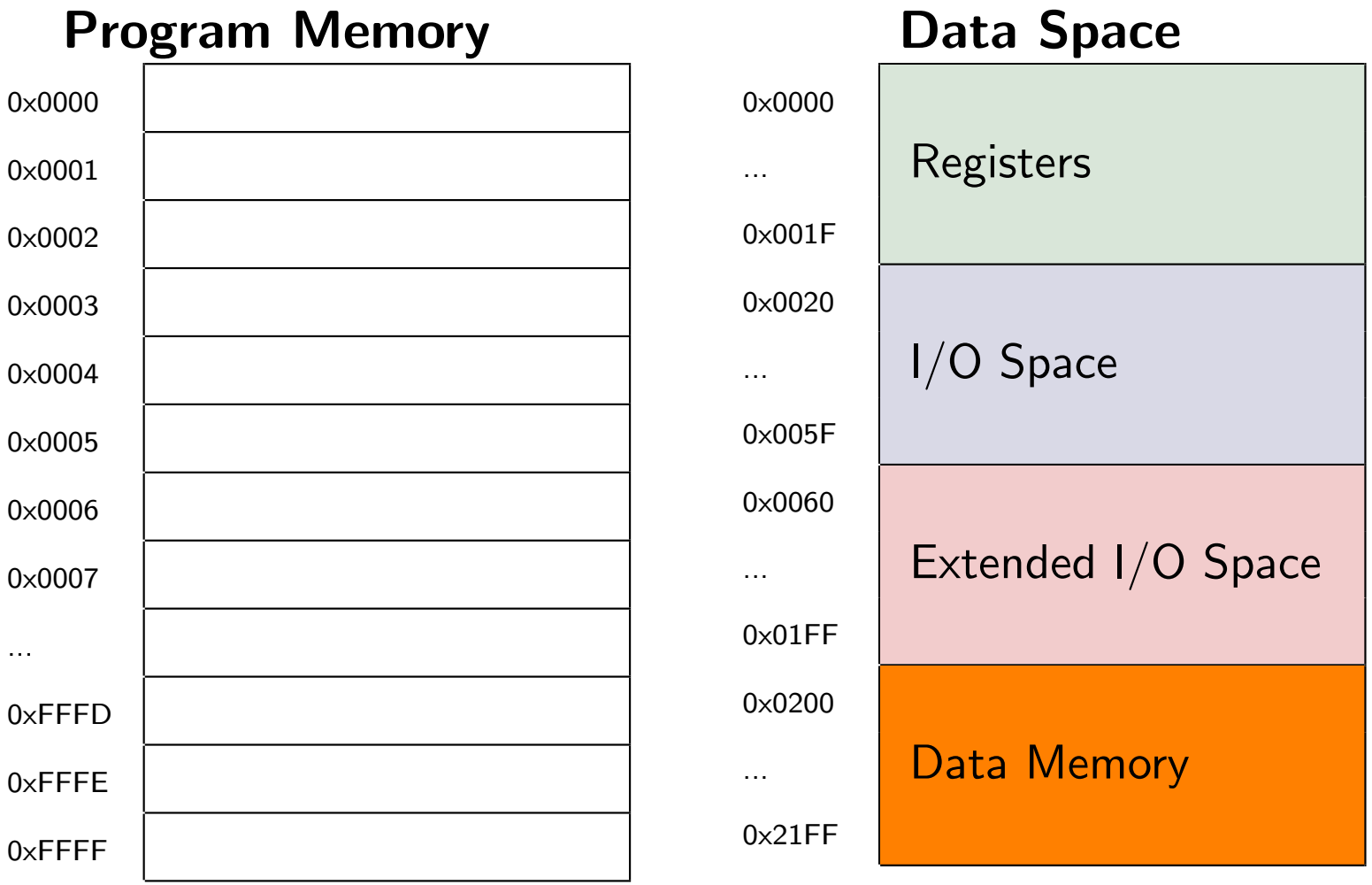
The actual size of the ATmega2560's program memory is 256KB ($256 \cdot 2^{10} = 262144$ bytes), giving a total of $128 \cdot 2^{10} = 131072$ addresses. The highest address is therefore $(131071)_{10} = \texttt{0x1ffff}$.

AVR Memory Buses (10)



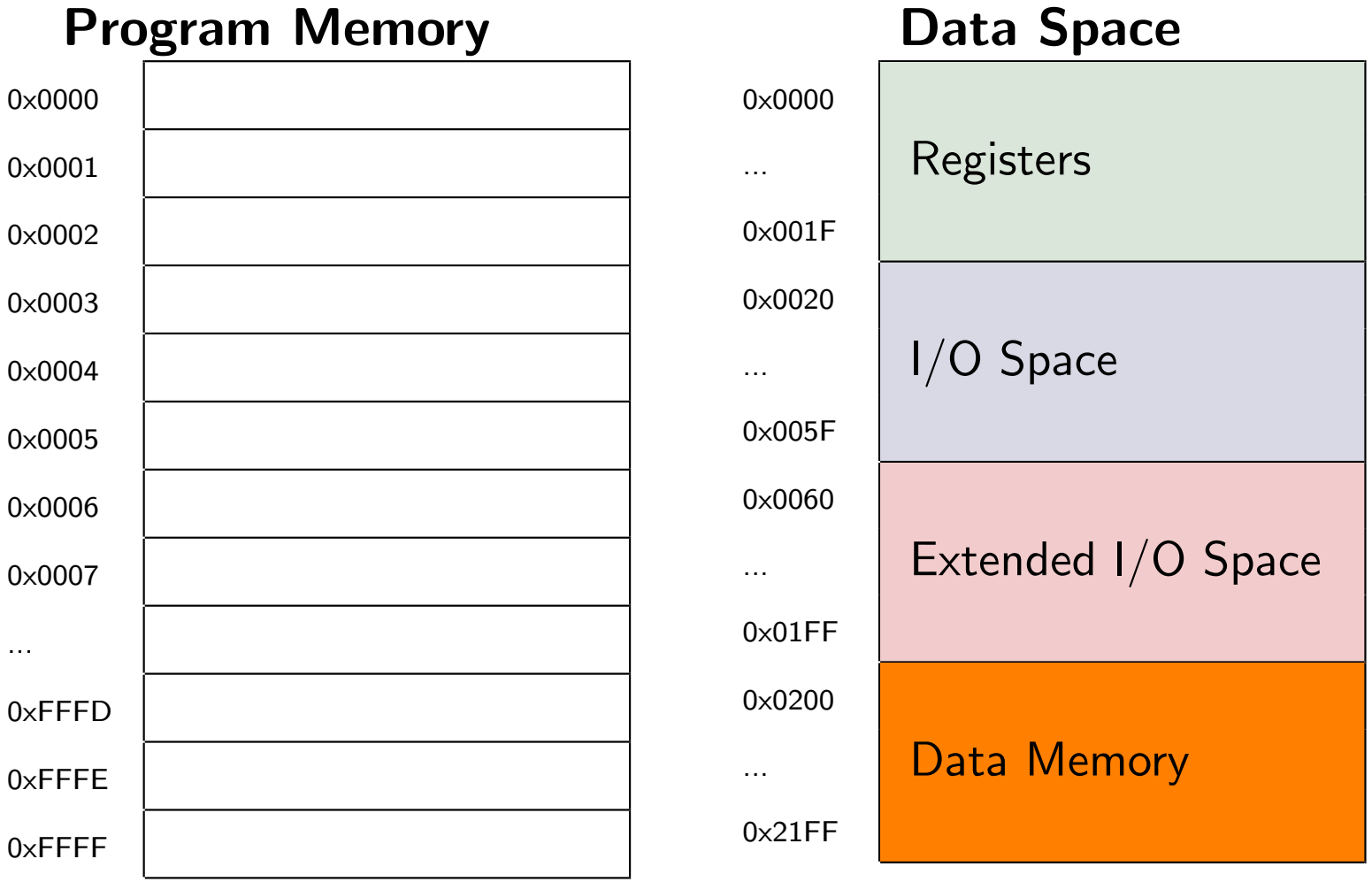
The size of the ATmega2560's data memory is 8KB ($8 \cdot 2^{10} = 8192$ bytes), giving a total of 0x2000 addressable locations. However, the first data memory address is not 0x0000.

Memory Spaces (1)



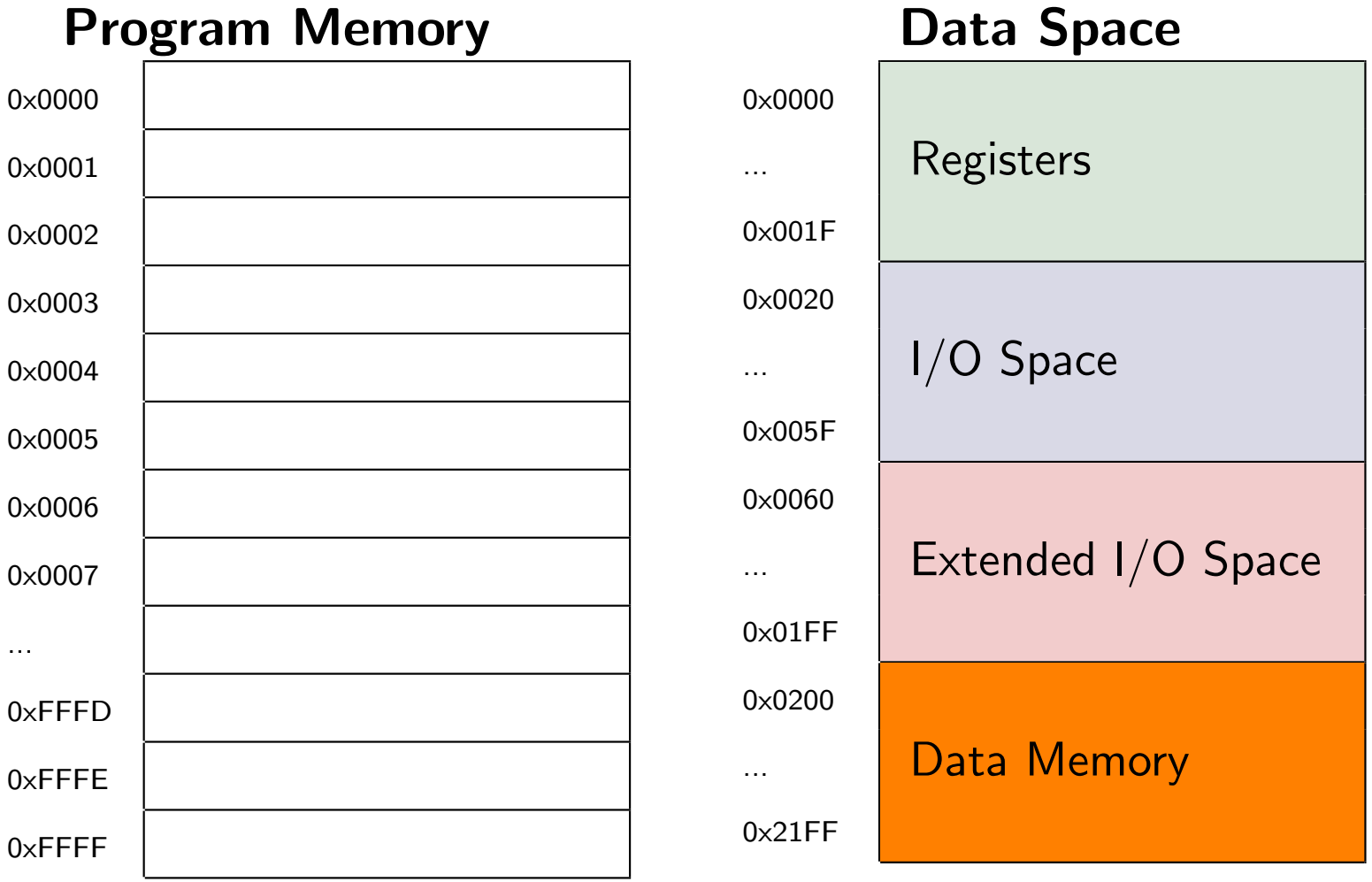
The 8KB of Data memory occupies the addresses from 0x0200 to 0x2200 in the data addressing space (or just 'data space').

Memory Spaces (2)



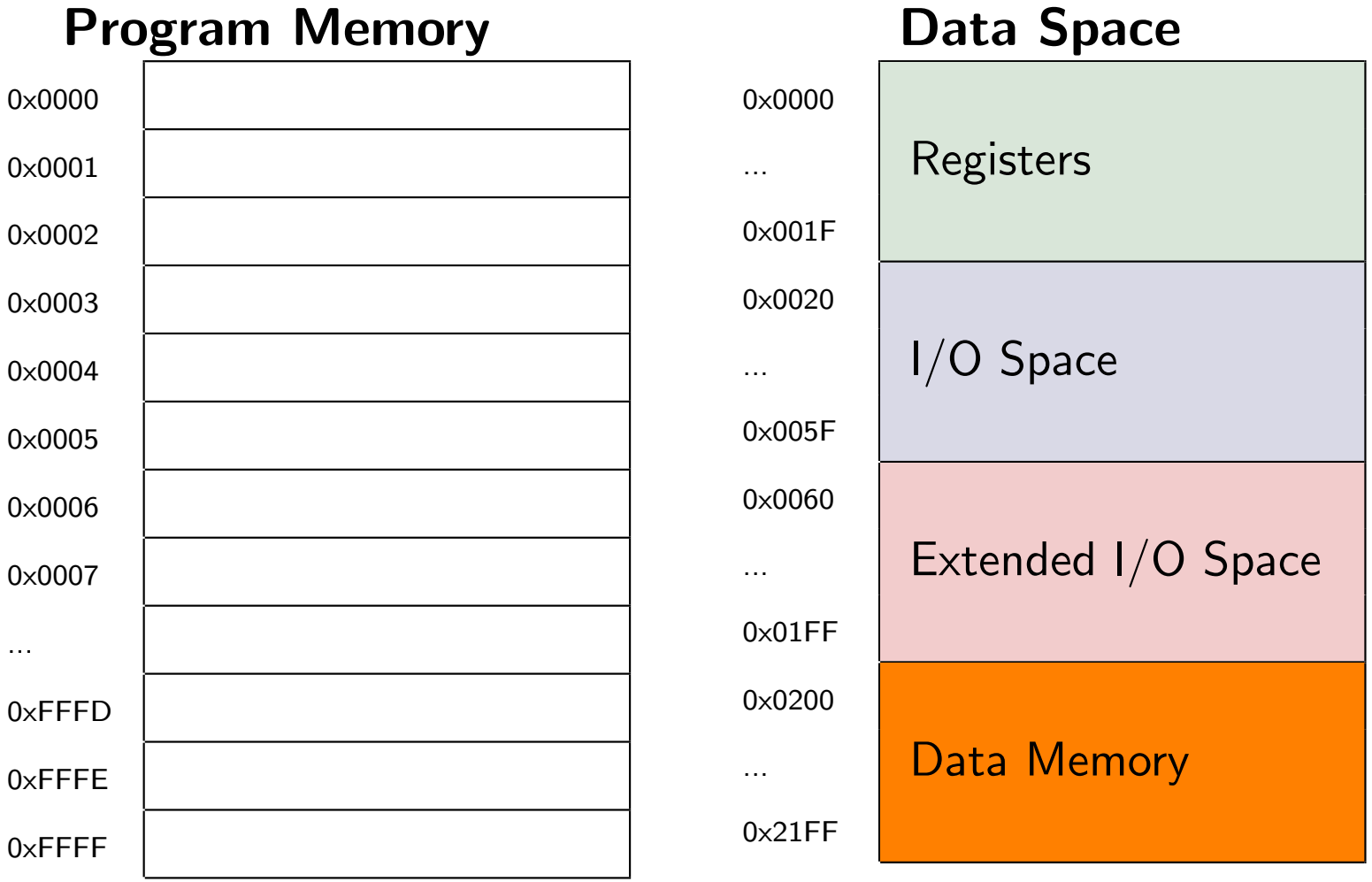
The addresses `0x0000` through `0x01ff` are used for other aspects of the system, but behaves like regular memory locations for the purpose of loading and storing their values.

Memory Spaces (3)



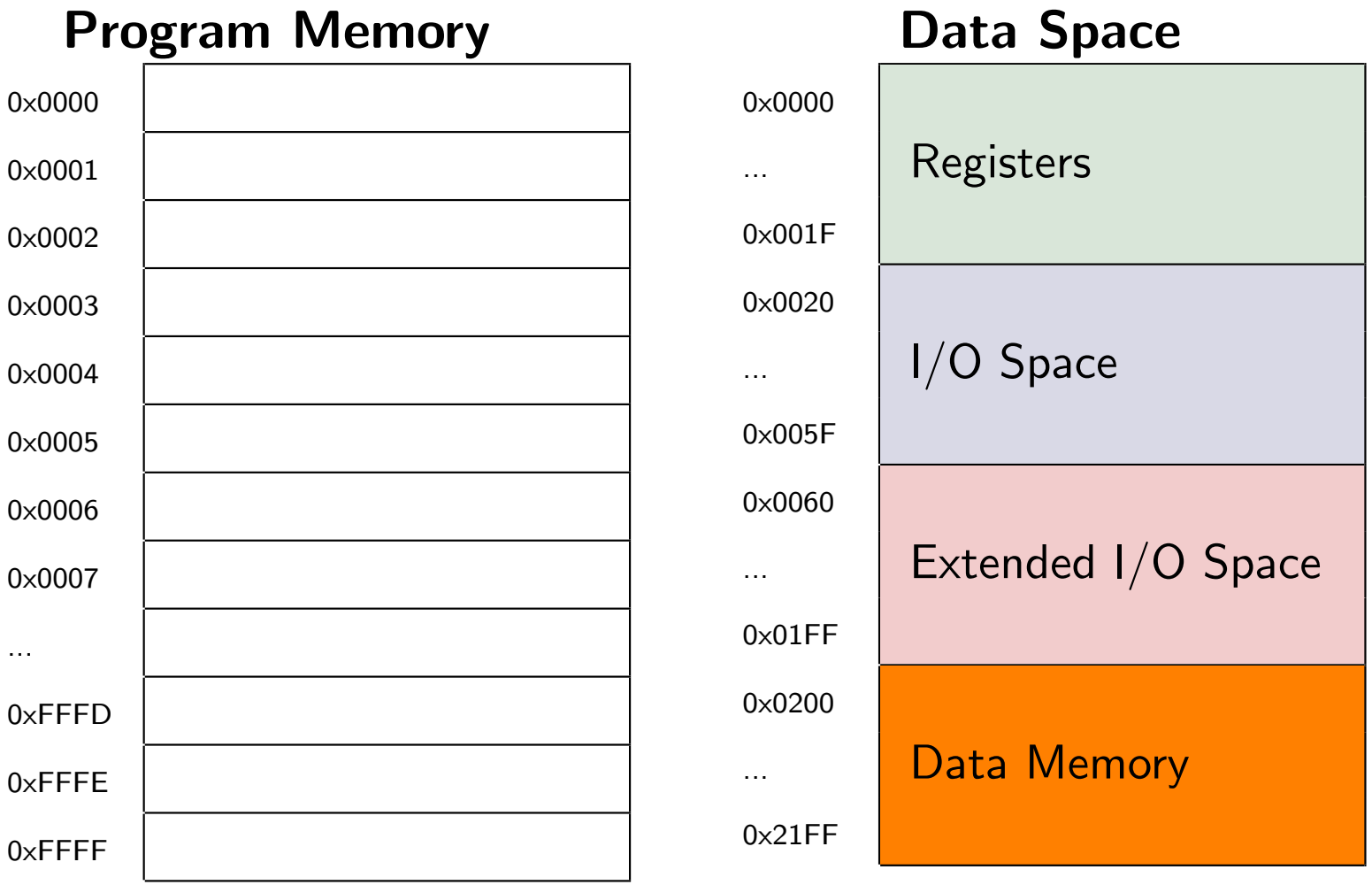
For example, the value of register 0 can be accessed at the data space address 0x0000, even though register 0 is an 'actual' memory location.

Memory Spaces (4)



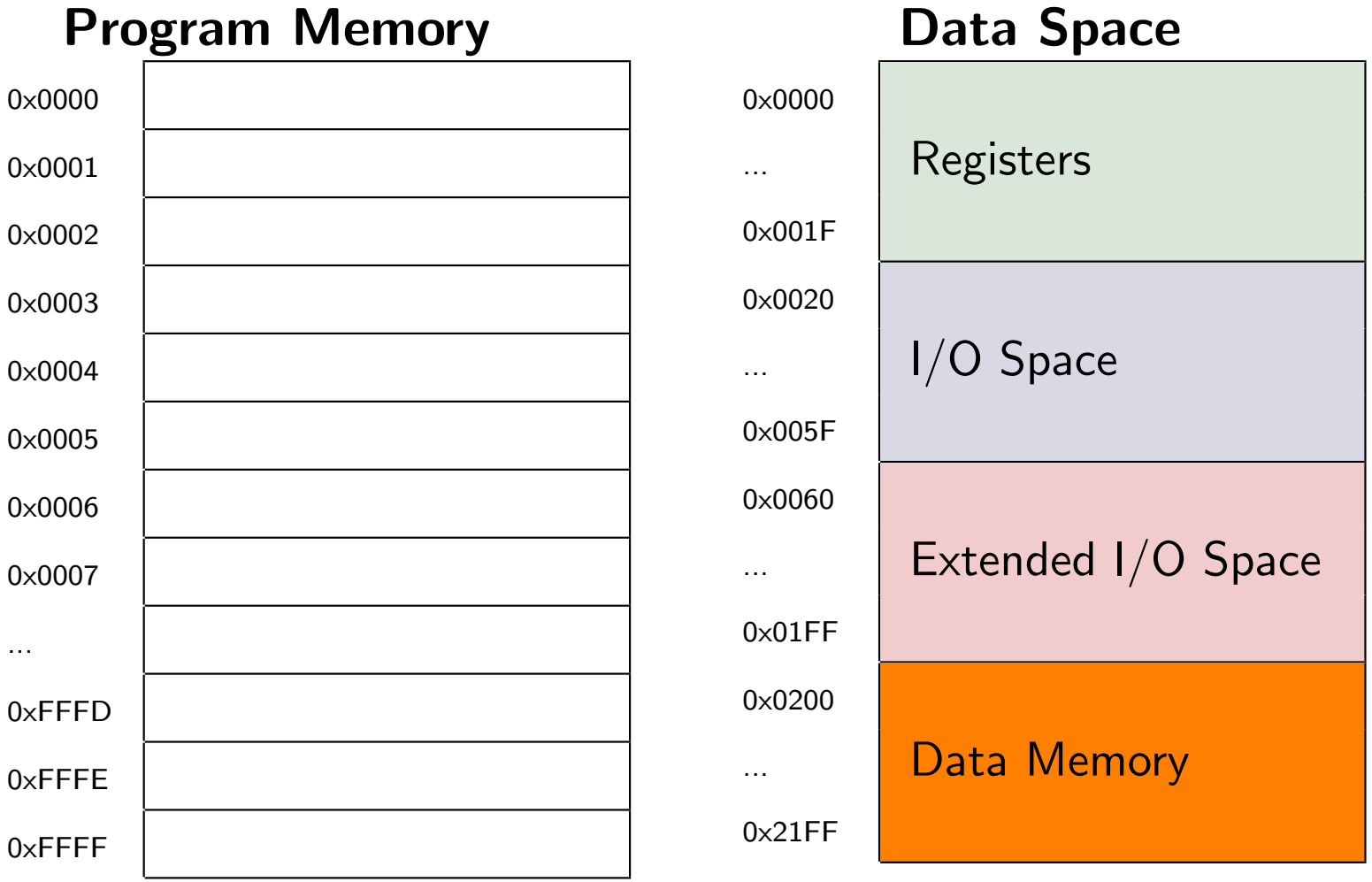
Similarly, the status register (SREG) can be directly manipulated at address `0x005f`.

Memory Spaces (5)



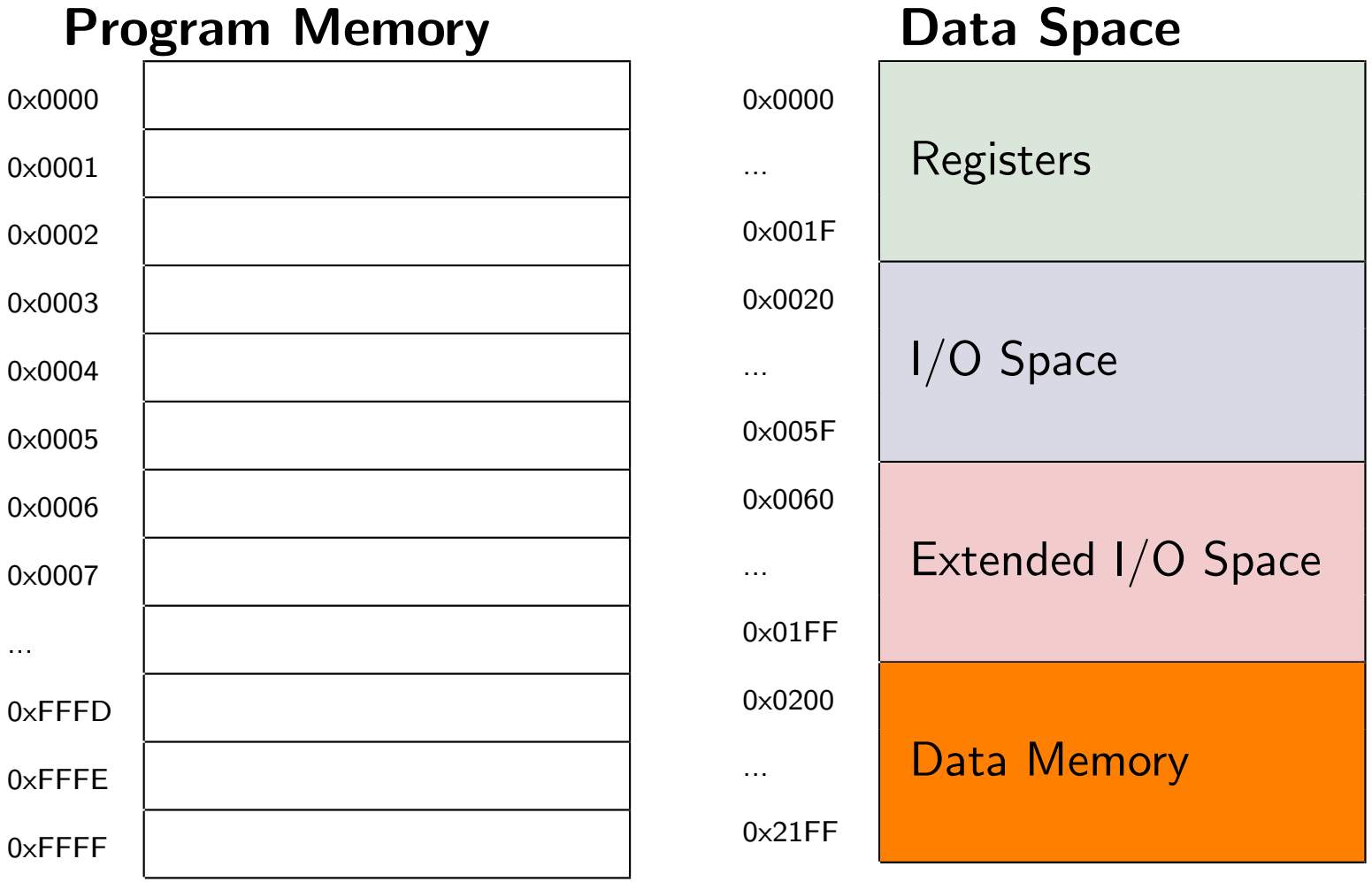
This behavior is called **memory mapping**, and provides a uniform mechanism for communicating with peripherals.

Memory Spaces (6)



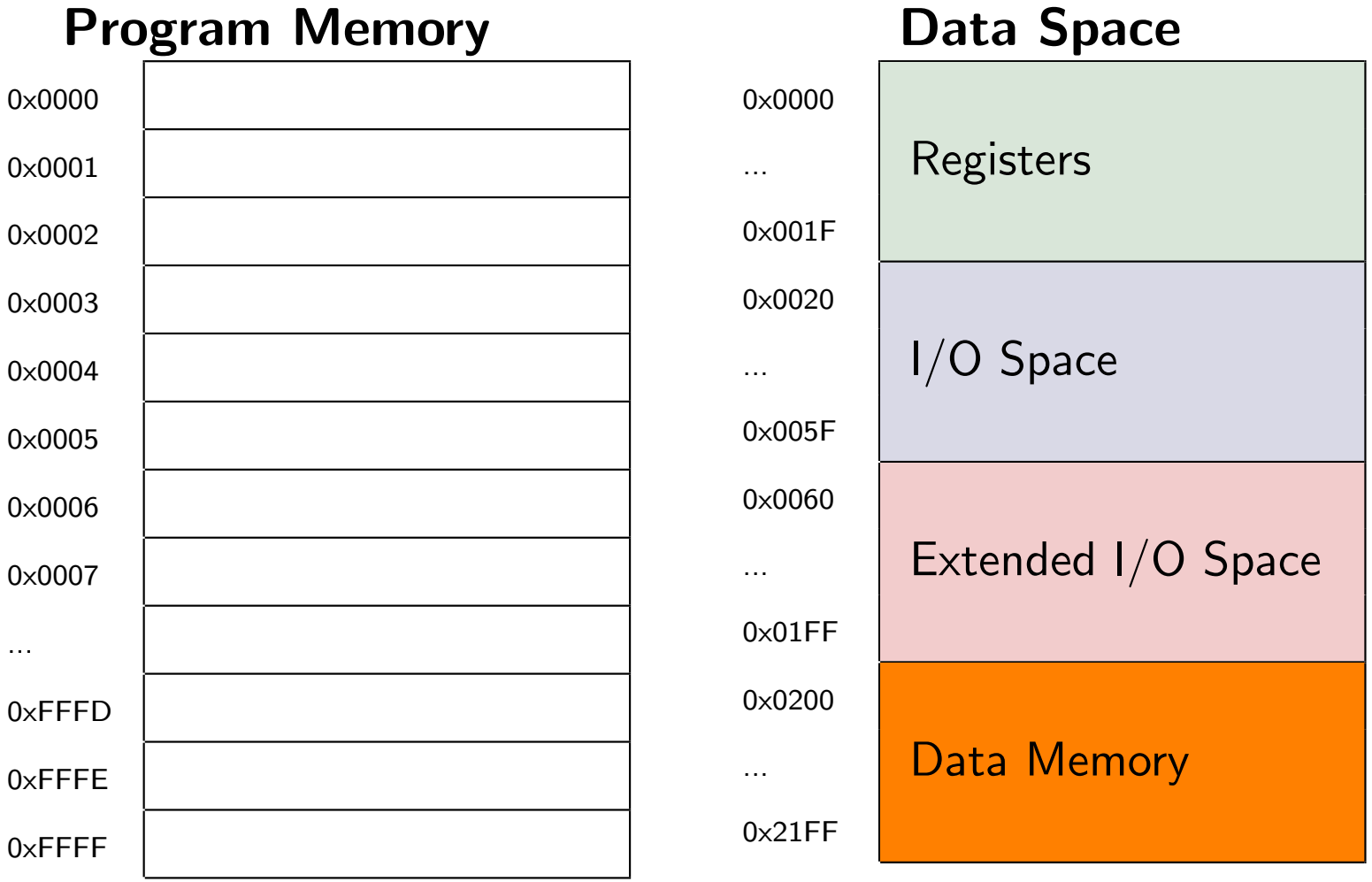
In a memory-mapped model, each peripheral's data or controls can be accessed or manipulated by reading/writing particular memory locations.

Memory Spaces (7)



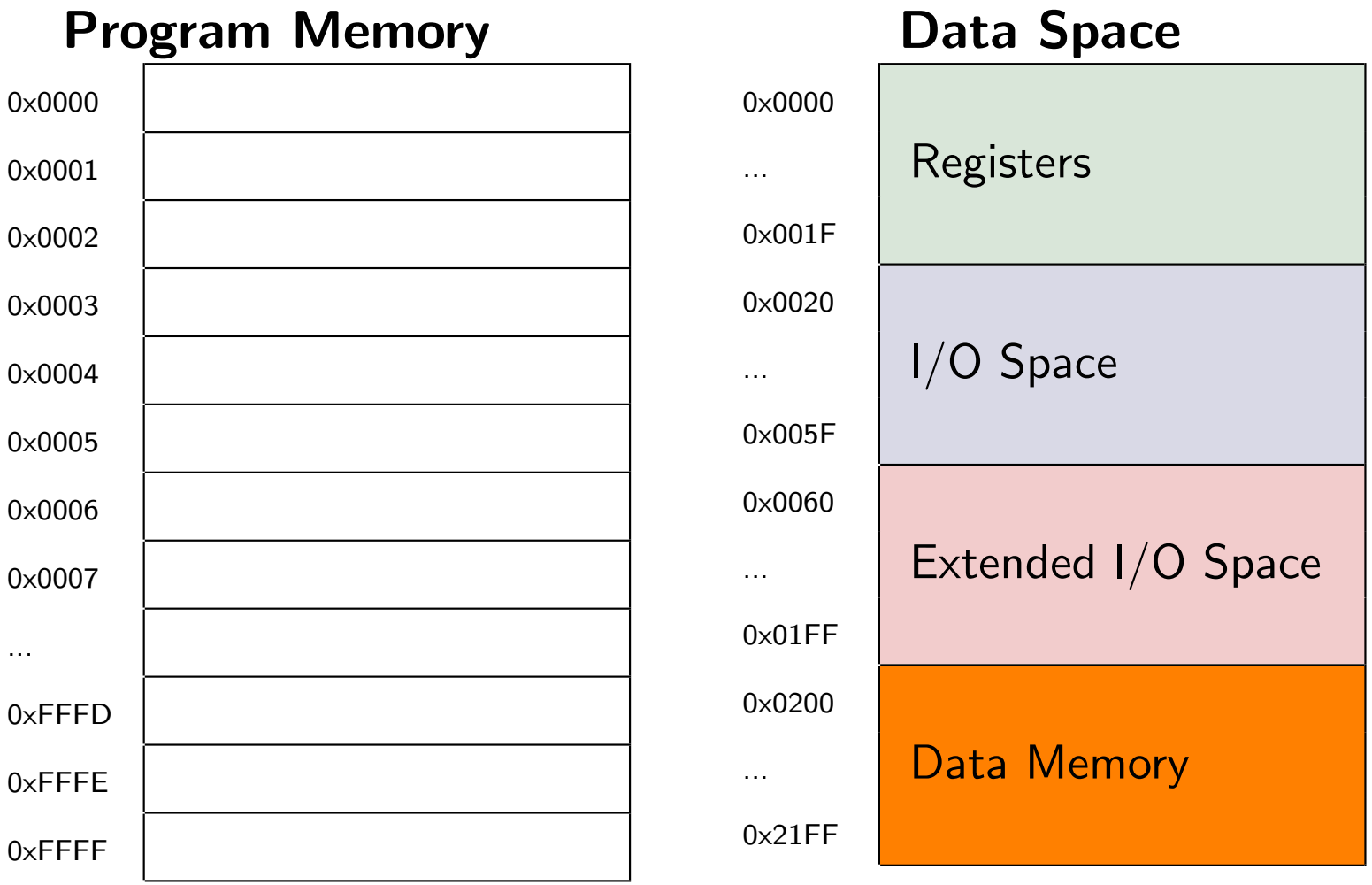
For example, the LED lights on the ATmega2560 board can be turned on/off by writing data to a particular location using `STS` instructions.

Memory Spaces (8)



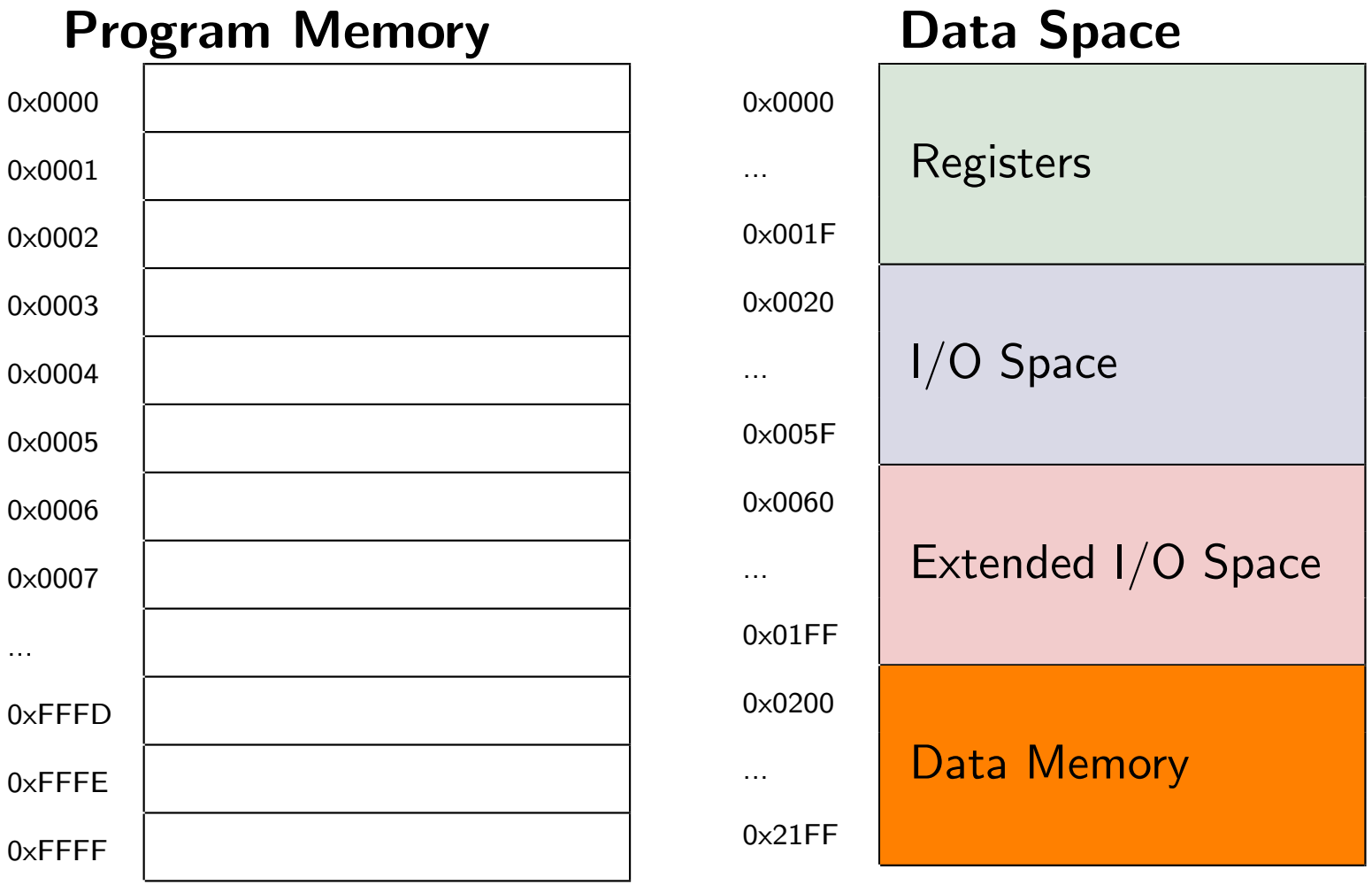
(The exact mechanism for controlling the LEDs will be covered in the labs.)

Memory Spaces (9)



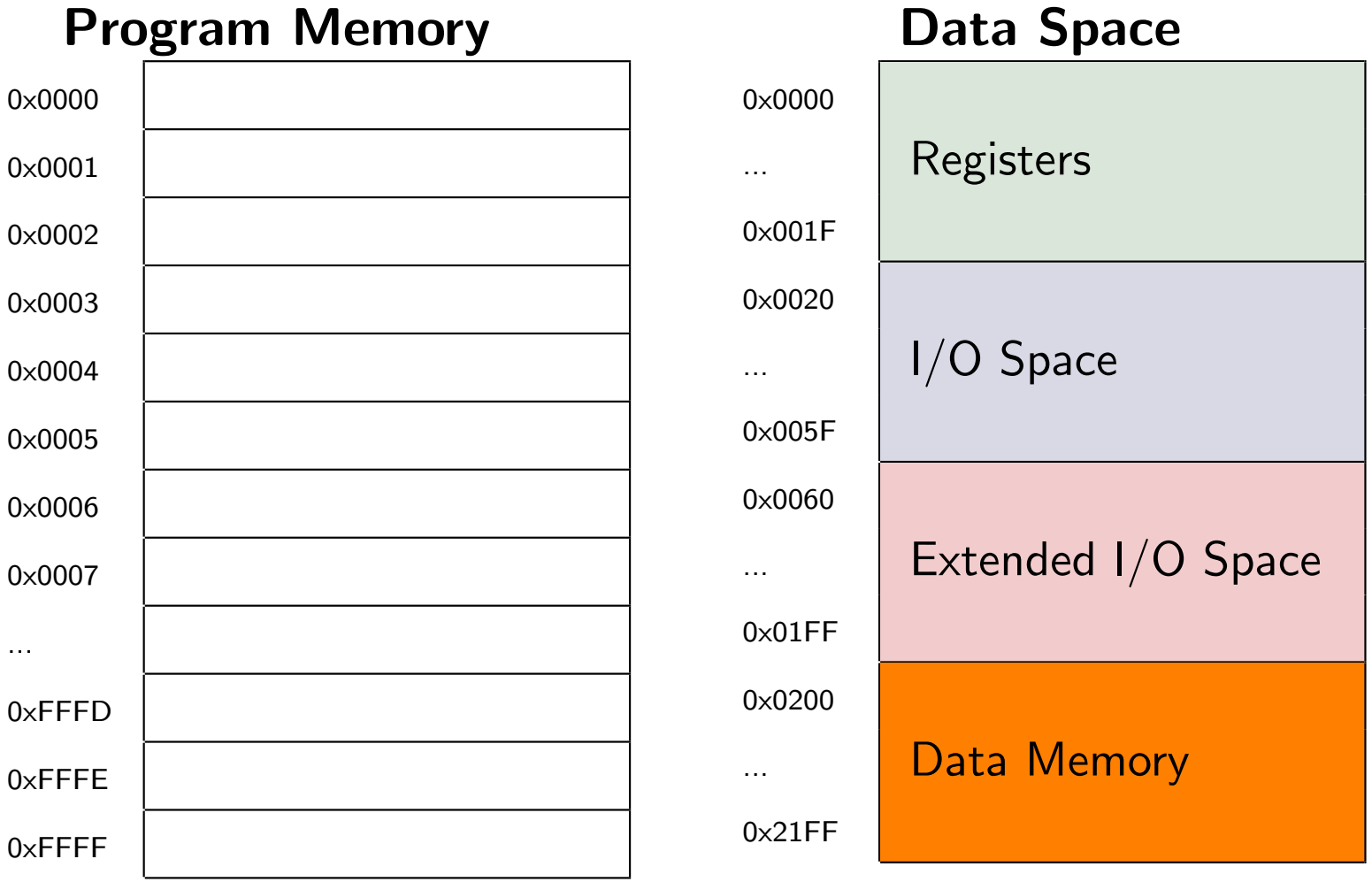
Addresses `0x0000` - `0x001f` refer to the 32 general registers `r0` - `r31`.

Memory Spaces (10)



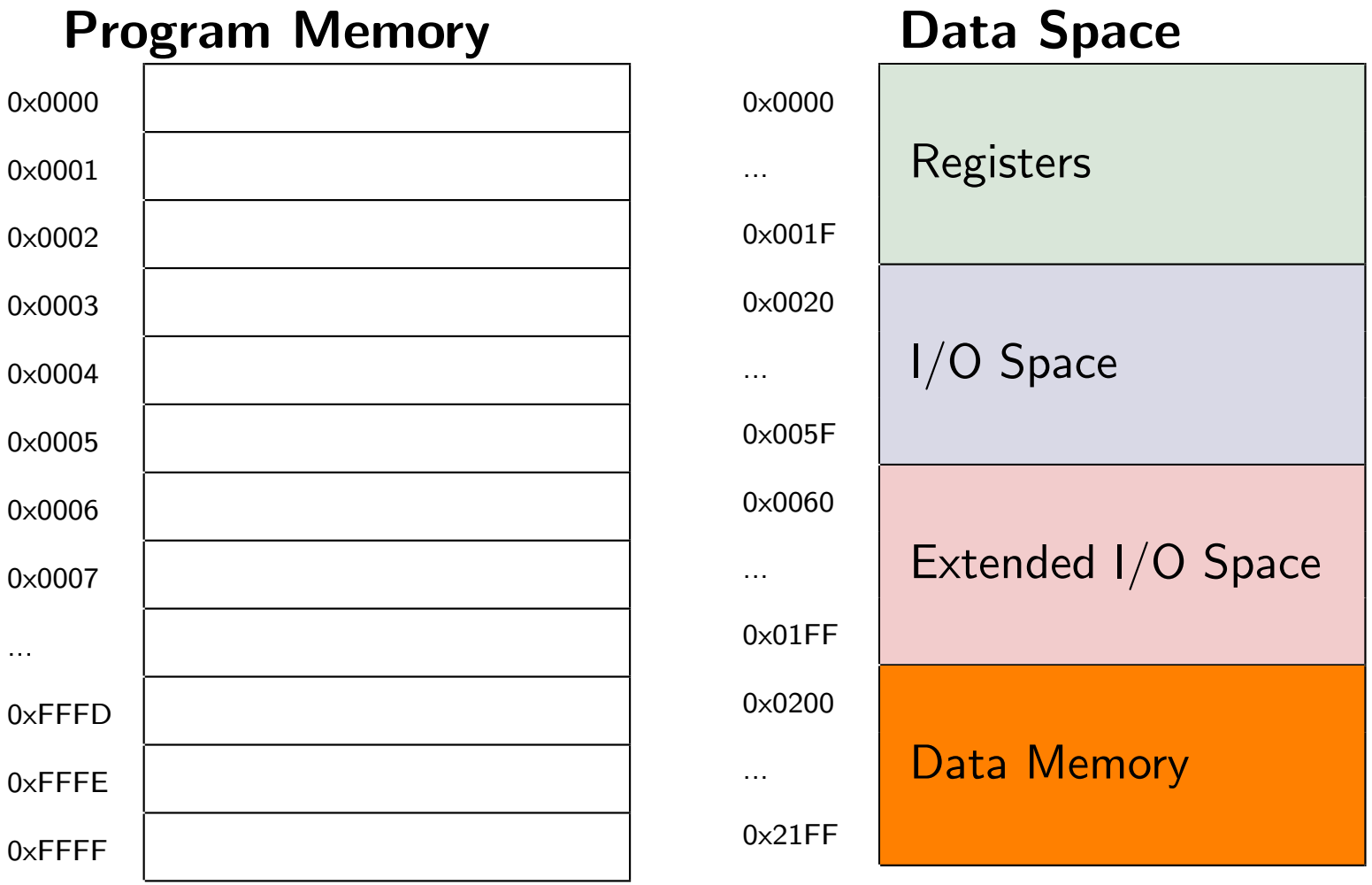
Addresses `0x0020` - `0x01ff` refer to 'I/O space', where the memory-mapped control registers for various peripherals and system data can be accessed or manipulated.

Memory Spaces (11)



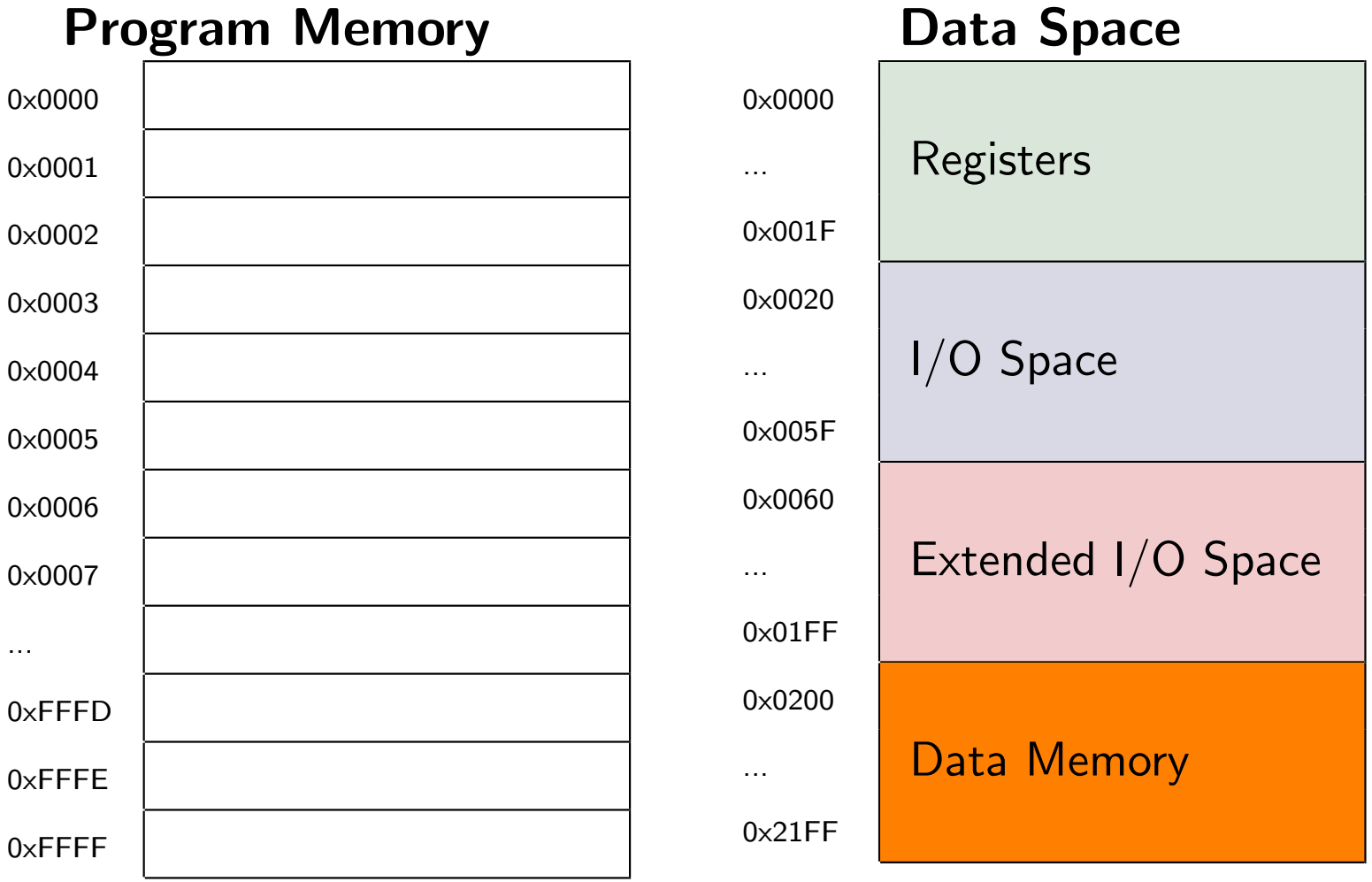
(Note that formally there is an 'I/O space' and an 'extended I/O space', but for our purposes both can be treated as a single block.)

Memory Spaces (12)



Finally, the actual data memory is available at addresses `0x0200` through `0x21FF`.

Memory Spaces (13)



The architecture allows additional external memory to be added starting at address 0x2200, but our boards do not have this feature.

Using Data Memory (1)

Program Memory

```
(0x0000)    ldi   r16, 0xe6
(0x0001)    sts   VAR1, r16
(0x0003)    lds   r17, VAR3
(0x0005)    inc   r17
(0x0006)    sts   VAR3, r17
(0x0008)    nop
            .dseg
            .org 0x200
            VAR1:   .byte 1
            VAR2:   .byte 1
            VAR3:   .byte 1
```

Registers

r16	0x00
r17	0x00
⋮	
r30 (ZL)	0x00
r31 (ZH)	0x00
r31:r30 (Z)	0x0000

C **N** **Z** **V**

Data Memory

Address	Contents
0x0200	0x00
0x0201	0x00
0x0202	0x00
0x0203	0x00
0x0204	0x00
0x0205	0x00
0x0206	0x00
0x0207	0x00
0x0208	0x00
0x0209	0x00

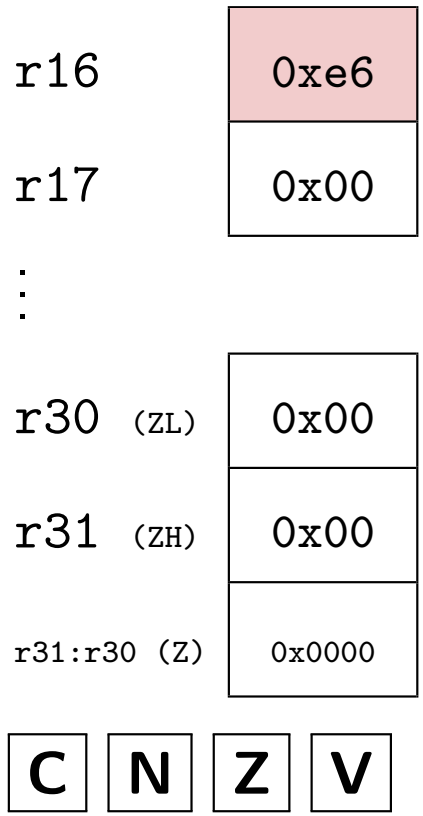
The primary memory access instructions are the `LD` (load) and `ST` (store) family. Although we have been using `LDI` to set the values of registers, it is not really a memory instruction (since it just sets a register to an immediate constant).

Using Data Memory (2)

Program Memory

```
(0x0000)    ldi   r16, 0xe6
(0x0001)    sts   VAR1, r16
(0x0003)    lds   r17, VAR3
(0x0005)    inc   r17
(0x0006)    sts   VAR3, r17
(0x0008)    nop
        .dseg
        .org 0x200
VAR1:   .byte 1
VAR2:   .byte 1
VAR3:   .byte 1
```

Registers



Data Memory

Address	Contents
0x0200	0x00
0x0201	0x00
0x0202	0x00
0x0203	0x00
0x0204	0x00
0x0205	0x00
0x0206	0x00
0x0207	0x00
0x0208	0x00
0x0209	0x00

The STS (store direct) instruction stores the value of a register to a fixed location in memory (the address of which has been hard-coded into the instruction).

Using Data Memory (3)

Program Memory

```
(0x0000)    ldi   r16, 0xe6
(0x0001)    sts   VAR1, r16
(0x0003)    lds   r17, VAR3
(0x0005)    inc   r17
(0x0006)    sts   VAR3, r17
(0x0008)    nop
        .dseg
        .org 0x200
VAR1:   .byte 1
VAR2:   .byte 1
VAR3:   .byte 1
```

Registers

r16	0xe6
r17	0x00
⋮	
r30 (ZL)	0x00
r31 (ZH)	0x00
r31:r30 (Z)	0x0000

C **N** **Z** **V**

Data Memory

Address	Contents
0x0200	0xe6
0x0201	0x00
0x0202	0x00
0x0203	0x00
0x0204	0x00
0x0205	0x00
0x0206	0x00
0x0207	0x00
0x0208	0x00
0x0209	0x00

Similarly, the LDS (load direct) loads the value of a hard-coded memory location into a register.

Using Data Memory (4)

Program Memory

```
(0x0000)    ldi   r16, 0xe6
(0x0001)    sts   VAR1, r16
(0x0003)    lds   r17, VAR3
(0x0005)    inc   r17
(0x0006)    sts   VAR3, r17
(0x0008)    nop
            .dseg
            .org 0x200
VAR1:       .byte 1
VAR2:       .byte 1
VAR3:       .byte 1
```

Registers

r16	0xe6
r17	0x01
⋮	
r30 (ZL)	0x00
r31 (ZH)	0x00
r31:r30 (Z)	0x0000

C **N** **Z** **V**

Data Memory

Address	Contents
0x0200	0xe6
0x0201	0x00
0x0202	0x01
0x0203	0x00
0x0204	0x00
0x0205	0x00
0x0206	0x00
0x0207	0x00
0x0208	0x00
0x0209	0x00

Note that, as usual, the labels (`VAR1` - `VAR3`) are conveniences: when the instructions are encoded, the label `VAR1` is replaced with its numerical value 0x200.

Using Data Memory (5)

Program Memory

```
(0x0000)    ldi   r16, 0xe6
(0x0001)    sts   VAR1, r16
(0x0003)    lds   r17, VAR3
(0x0005)    inc   r17
(0x0006)    sts   VAR3, r17
(0x0008)    nop
            .dseg
            .org 0x200
VAR1:       .byte 1
VAR2:       .byte 1
VAR3:       .byte 1
```

Registers

r16	0xe6
r17	0x01
⋮	
r30 (ZL)	0x00
r31 (ZH)	0x00
r31:r30 (Z)	0x0000

C **N** **Z** **V**

Data Memory

Address	Contents
0x0200	0xe6
0x0201	0x00
0x0202	0x01
0x0203	0x00
0x0204	0x00
0x0205	0x00
0x0206	0x00
0x0207	0x00
0x0208	0x00
0x0209	0x00

The LDS and STS instructions use **direct addressing**, and since the address is hard coded into the instruction, these instructions are not suitable for cases where the desired address may change (such as array indexing).

Using Data Memory (6)

Program Memory

```
(0x0000)    ldi   r16, 0xe6
(0x0001)    sts   VAR1, r16
(0x0003)    lds   r17, VAR3
(0x0005)    inc   r17
(0x0006)    sts   VAR3, r17
(0x0008)    nop
            .dseg
            .org 0x200
VAR1:       .byte 1
VAR2:       .byte 1
VAR3:       .byte 1
```

Registers

r16	0xe6
r17	0x01
⋮	
r30 (ZL)	0x00
r31 (ZH)	0x00
r31:r30 (Z)	0x0000

C **N** **Z** **V**

Data Memory

Address	Contents
0x0200	0xe6
0x0201	0x00
0x0202	0x01
0x0203	0x00
0x0204	0x00
0x0205	0x00
0x0206	0x00
0x0207	0x00
0x0208	0x00
0x0209	0x00

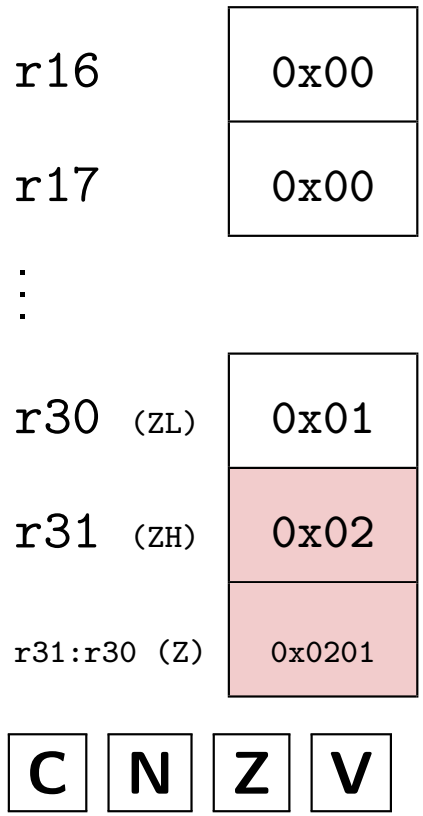
(In theory, the address could be modified at run time using self-modifying code, but this is generally not advisable on Harvard architectures and would dramatically reduce the lifespan of the AVR board's flash memory.)

Using Data Memory (7)

Program Memory

```
(0x0000)    ldi   r30, low(VAR2)
(0x0001)    ldi   r31, high(VAR2)
(0x0002)    ldi   r16, 0xe6
(0x0003)    st    Z, r16
(0x0004)    nop
        .dseg
        .org 0x200
VAR1:   .byte 1
VAR2:   .byte 1
```

Registers



Data Memory

Address	Contents
0x0200	0x00
0x0201	0x00
0x0202	0x00
0x0203	0x00
0x0204	0x00
0x0205	0x00
0x0206	0x00
0x0207	0x00
0x0208	0x00
0x0209	0x00

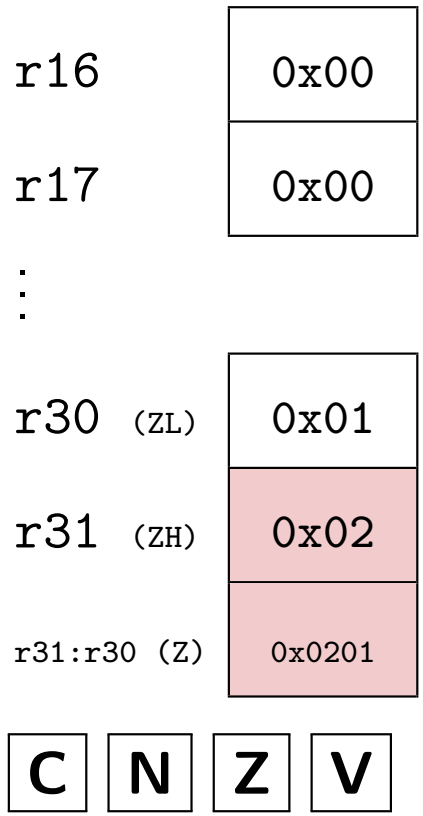
The LD and ST instructions use **indirect addressing**, where the address of the load/store is taken from registers instead of being hard-coded.

Using Data Memory (8)

Program Memory

```
(0x0000)    ldi   r30, low(VAR2)
(0x0001)    ldi   r31, high(VAR2)
(0x0002)    ldi   r16, 0xe6
(0x0003)    st    Z, r16
(0x0004)    nop
        .dseg
        .org 0x200
VAR1:   .byte 1
VAR2:   .byte 1
```

Registers



Data Memory

Address	Contents
0x0200	0x00
0x0201	0x00
0x0202	0x00
0x0203	0x00
0x0204	0x00
0x0205	0x00
0x0206	0x00
0x0207	0x00
0x0208	0x00
0x0209	0x00

Since memory addresses are 16 bits long, two registers are needed to store a memory address. The LD and ST instructions will only work with three particular register pairs.

Using Data Memory (9)

Program Memory

```
(0x0000)    ldi   r30, low(VAR2)
(0x0001)    ldi   r31, high(VAR2)
(0x0002)    ldi   r16, 0xe6
(0x0003)    st    Z, r16
(0x0004)    nop
            .dseg
            .org 0x200
VAR1:       .byte 1
VAR2:       .byte 1
```

Registers

r16	0x00
r17	0x00
⋮	
r30 (ZL)	0x01
r31 (ZH)	0x02
r31:r30 (Z)	0x0201

C **N** **Z** **V**

Data Memory

Address	Contents
0x0200	0x00
0x0201	0x00
0x0202	0x00
0x0203	0x00
0x0204	0x00
0x0205	0x00
0x0206	0x00
0x0207	0x00
0x0208	0x00
0x0209	0x00

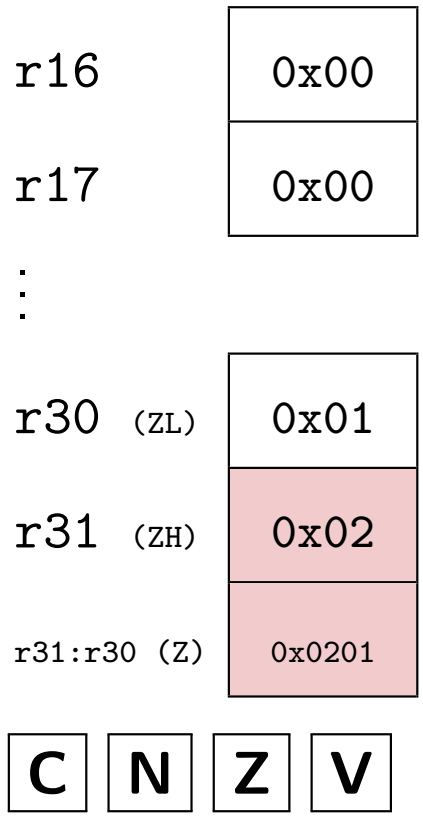
The `X` (`r27:r26`), `X` (`r29:r28`) and `X` (`r31:r30`) pairs are used to store 16-bit addresses (in little endian order) for the indirect load/-store instructions.

Using Data Memory (10)

Program Memory

```
(0x0000)    ldi   r30, low(VAR2)
(0x0001)    ldi   r31, high(VAR2)
(0x0002)    ldi   r16, 0xe6
(0x0003)    st    Z, r16
(0x0004)    nop
        .dseg
        .org 0x200
VAR1:   .byte 1
VAR2:   .byte 1
```

Registers



Data Memory

Address	Contents
0x0200	0x00
0x0201	0x00
0x0202	0x00
0x0203	0x00
0x0204	0x00
0x0205	0x00
0x0206	0x00
0x0207	0x00
0x0208	0x00
0x0209	0x00

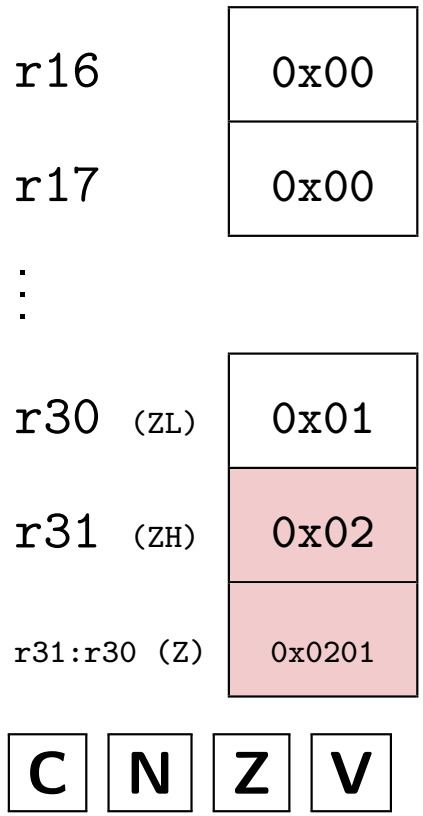
Some instructions (such as `LPM` and some encodings of `LD`/`ST`) only allow `Z` or `Y`/`Z`. See the documentation for more details.

Using Data Memory (11)

Program Memory

```
(0x0000)    ldi   r30, low(VAR2)
(0x0001)    ldi   r31, high(VAR2)
(0x0002)    ldi   r16, 0xe6
(0x0003)    st    Z, r16
(0x0004)    nop
        .dseg
        .org 0x200
VAR1:   .byte 1
VAR2:   .byte 1
```

Registers



Data Memory

Address	Contents
0x0200	0x00
0x0201	0x00
0x0202	0x00
0x0203	0x00
0x0204	0x00
0x0205	0x00
0x0206	0x00
0x0207	0x00
0x0208	0x00
0x0209	0x00

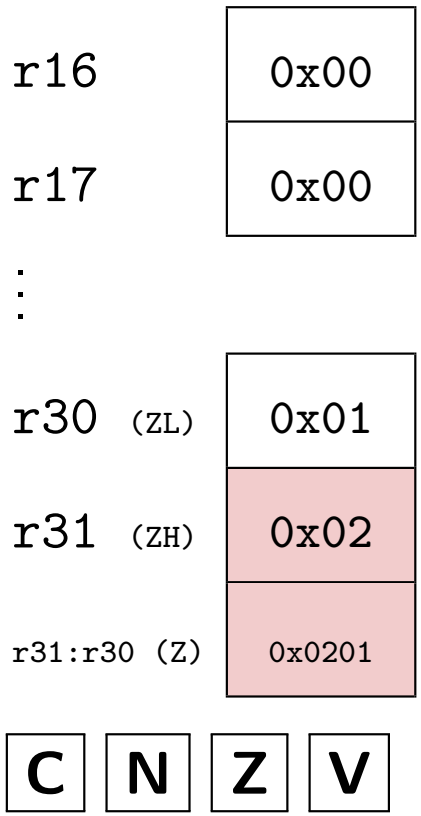
In the example above, the 16-bit address of the label `VAR2` is loaded into the Z pair using the `low` and `high` macros to load the low and high bytes into `r31` (ZL) and `r30` (ZH), respectively.

Using Data Memory (12)

Program Memory

```
(0x0000)    ldi   ZL, low(VAR2)
(0x0001)    ldi   ZH, high(VAR2)
(0x0002)    ldi   r16, 0xe6
(0x0003)    st    Z, r16
(0x0004)    nop
        .dseg
        .org 0x200
VAR1:   .byte 1
VAR2:   .byte 1
```

Registers



Data Memory

Address	Contents
0x0200	0x00
0x0201	0x00
0x0202	0x00
0x0203	0x00
0x0204	0x00
0x0205	0x00
0x0206	0x00
0x0207	0x00
0x0208	0x00
0x0209	0x00

Normally, the names ZL and ZH (or similarly, XL/XH and YL/YH) are defined as aliases of r31 and r30, so they can be used directly.

Using Data Memory (13)

Program Memory

```
(0x0000)    ldi   ZL, low(VAR2)
(0x0001)    ldi   ZH, high(VAR2)
(0x0002)    ldi   r16, 0xe6
(0x0003)    st    Z, r16
(0x0004)    nop
            .dseg
            .org 0x200
VAR1:       .byte 1
VAR2:       .byte 1
```

Registers

r16	0xe6
r17	0x00
⋮	
r30 (ZL)	0x01
r31 (ZH)	0x02
r31:r30 (Z)	0x0201

C **N** **Z** **V**

Data Memory

Address	Contents
0x0200	0x00
0x0201	0xe6
0x0202	0x00
0x0203	0x00
0x0204	0x00
0x0205	0x00
0x0206	0x00
0x0207	0x00
0x0208	0x00
0x0209	0x00

The indirect ST instruction then stores the value 0xe6 into the address contained in the Z register (0x0201, the address of VAR2).

Addressing Modes (1)

Mode	**Result**
Immediate	C
Direct	$[A]$
Indirect	$[X]$
Indirect with Displacement	$[X + C]$

C = Constant Operand, A = Constant Address

X = Register Pair (e.g. X, Y or Z)

The addressing modes supported by the AVR architecture are summarized with pseudocode in the table above. The notation $[A]$ is used to mean 'the contents of memory at address A'. Similar notation (with round brackets) is used in the documentation.

Addressing Modes (2)

Mode	**Result**
Immediate	C
Direct	$[A]$
Indirect	$[X]$
Indirect with Displacement	$[X + C]$

C = Constant Operand, A = Constant Address

X = Register Pair (e.g. X, Y or Z)

The `LDI` instruction is not really a load instruction, but can be considered to be an 'immediate' addressing mode.

Addressing Modes (3)

Mode	**Result**
Immediate	C
Direct	$[A]$
Indirect	$[X]$
Indirect with Displacement	$[X + C]$

C = Constant Operand, A = Constant Address

X = Register Pair (e.g. X, Y or Z)

The `LD` and `ST` instructions implement the indirect addressing mode. The displaced indirect mode is implemented by the `LDD` and `STD` instructions.

Addressing Modes (4)

Mode	**Result**
Immediate	C
Direct	$[A]$
Indirect	$[X]$
Indirect with Displacement	$[X + C]$

C = Constant Operand, A = Constant Address

X = Register Pair (e.g. X, Y or Z)

When we come back to the general topic of architecture (in July), we will see that this set of addressing modes is relatively small compared to many other architectures.

Arrays (1)

Program Memory

```
(0x0000)    ldi   ZL, low(ARR)
(0x0001)    ldi   ZH, high(ARR)
(0x0002)    ldi   r16, 0xe1
(0x0003)    st    Z, r16
(0x0004)    adiw  ZL:ZH, 1
(0x0005)    ldi   r17, 0xe6
(0x0006)    st    Z, r17
(0x0007)    nop
        .dseg
        .org 0x200
        ARR: .byte 5
```

Registers

r16	0x00
r17	0x00
⋮	
r30 (ZL)	0x00
r31 (ZH)	0x00
r31:r30 (Z)	0x0000

C **N** **Z** **V**

Data Memory

Address	Contents
0x0200	0x00
0x0201	0x00
0x0202	0x00
0x0203	0x00
0x0204	0x00
0x0205	0x00
0x0206	0x00
0x0207	0x00
0x0208	0x00
0x0209	0x00

An array is just a contiguous block of memory locations. The block ARR defined above consists of 5 bytes of space at address 0x200.

Arrays (2)

Program Memory

```
(0x0000)    ldi   ZL, low(ARR)
(0x0001)    ldi   ZH, high(ARR)
(0x0002)    ldi   r16, 0xe1
(0x0003)    st    Z, r16
(0x0004)    adiw  ZL:ZH, 1
(0x0005)    ldi   r17, 0xe6
(0x0006)    st    Z, r17
(0x0007)    nop
        .dseg
        .org 0x200
        ARR: .byte 5
```

Registers

r16	0xe1
r17	0x00
⋮	
r30 (ZL)	0x00
r31 (ZH)	0x02
r31:r30 (Z)	0x0200

C **N** **Z** **V**

Data Memory

Address	Contents
0x0200	0xe1
0x0201	0x00
0x0202	0x00
0x0203	0x00
0x0204	0x00
0x0205	0x00
0x0206	0x00
0x0207	0x00
0x0208	0x00
0x0209	0x00

When a block of memory is used as an array, the address of the beginning of the block is often called the **base address**. The address of each element can be computed by adding an index (or **offset**) to the base address.

Arrays (3)

Program Memory

```
(0x0000)    ldi   ZL, low(ARR)
(0x0001)    ldi   ZH, high(ARR)
(0x0002)    ldi   r16, 0xe1
(0x0003)    st    Z, r16
(0x0004)    adiw  ZL:ZH, 1
(0x0005)    ldi   r17, 0xe6
(0x0006)    st    Z, r17
(0x0007)    nop
        .dseg
        .org 0x200
        ARR: .byte 5
```

Registers

r16	0xe1
r17	0x00
⋮	
r30 (ZL)	0x00
r31 (ZH)	0x02
r31:r30 (Z)	0x0200

C **N** **Z** **V**

Data Memory

Address	Contents
0x0200	0xe1
0x0201	0x00
0x0202	0x00
0x0203	0x00
0x0204	0x00
0x0205	0x00
0x0206	0x00
0x0207	0x00
0x0208	0x00
0x0209	0x00

After the first ST instruction above, the first element of ARR at index 0 (address 0x200 + 0) is set to 0xe1.

Arrays (4)

Program Memory

```
(0x0000)    ldi   ZL, low(ARR)
(0x0001)    ldi   ZH, high(ARR)
(0x0002)    ldi   r16, 0xe1
(0x0003)    st    Z, r16
(0x0004)    adiw  ZL:ZH, 1
(0x0005)    ldi   r17, 0xe6
(0x0006)    st    Z, r17
(0x0007)    nop
        .dseg
        .org 0x200
        ARR: .byte 5
```

Registers

r16	0xe1
r17	0x00
⋮	
r30 (ZL)	0x01
r31 (ZH)	0x02
r31:r30 (Z)	0x0201

C **N** **Z** **V**

Data Memory

Address	Contents
0x0200	0xe1
0x0201	0x00
0x0202	0x00
0x0203	0x00
0x0204	0x00
0x0205	0x00
0x0206	0x00
0x0207	0x00
0x0208	0x00
0x0209	0x00

The `ADIW` instruction can be used to add constants to 16-bit register pairs directly (but the addition can also be performed with `ADD` and `ADC` as usual).

Arrays (5)

Program Memory

```
(0x0000)    ldi   ZL, low(ARR)
(0x0001)    ldi   ZH, high(ARR)
(0x0002)    ldi   r16, 0xe1
(0x0003)    st    Z, r16
(0x0004)    adiw  ZL:ZH, 1
(0x0005)    ldi   r17, 0xe6
(0x0006)    st    Z, r17
(0x0007)    nop
        .dseg
        .org 0x200
        ARR: .byte 5
```

Registers

r16	0xe1
r17	0x00
⋮	
r30 (ZL)	0x01
r31 (ZH)	0x02
r31:r30 (Z)	0x0201

C **N** **Z** **V**

Data Memory

Address	Contents
0x0200	0xe1
0x0201	0x00
0x0202	0x00
0x0203	0x00
0x0204	0x00
0x0205	0x00
0x0206	0x00
0x0207	0x00
0x0208	0x00
0x0209	0x00

After the ADIW, the Z register pair contains the address of index 1 of ARR.

Arrays (6)

Program Memory

```
(0x0000)    ldi   ZL, low(ARR)
(0x0001)    ldi   ZH, high(ARR)
(0x0002)    ldi   r16, 0xe1
(0x0003)    st    Z, r16
(0x0004)    adiw  ZL:ZH, 1
(0x0005)    ldi   r17, 0xe6
(0x0006)    st    Z, r17
(0x0007)    nop
        .dseg
        .org 0x200
        ARR: .byte 5
```

Registers

r16	0xe1
r17	0xe6
⋮	
r30 (ZL)	0x01
r31 (ZH)	0x02
r31:r30 (Z)	0x0201

C **N** **Z** **V**

Data Memory

Address	Contents
0x0200	0xe1
0x0201	0xe6
0x0202	0x00
0x0203	0x00
0x0204	0x00
0x0205	0x00
0x0206	0x00
0x0207	0x00
0x0208	0x00
0x0209	0x00

Manipulating the numerical value of Z allows different indices of the array to be accessed or modified by LD or ST.

Arrays (7)

Program Memory

```
(0x0000)    ldi   ZL, low(ARR)
(0x0001)    ldi   ZH, high(ARR)
(0x0002)    ldi   r16, 0xe1
(0x0003)    ldi   r17, 0xe6
(0x0004)    st    Z+, r16
(0x0005)    st    Z+, r17
(0x0006)    st    Z+, r16
(0x0007)    nop
        .dseg
        .org 0x200
        ARR: .byte 5
```

Registers

r16	0xe1
r17	0xe6
⋮	
r30 (ZL)	0x00
r31 (ZH)	0x02
r31:r30 (Z)	0x0200

C **N** **Z** **V**

Data Memory

Address	Contents
0x0200	0x00
0x0201	0x00
0x0202	0x00
0x0203	0x00
0x0204	0x00
0x0205	0x00
0x0206	0x00
0x0207	0x00
0x0208	0x00
0x0209	0x00

The LD and ST instructions also offer 'indirect with post-increment/pre-decrement' encodings. The 'post-increment' variant (shown above) automatically increments the value of the address **after** performing the load/store.

Arrays (8)

Program Memory

```
(0x0000)    ldi   ZL, low(ARR)
(0x0001)    ldi   ZH, high(ARR)
(0x0002)    ldi   r16, 0xe1
(0x0003)    ldi   r17, 0xe6
(0x0004)    st    Z+, r16
(0x0005)    st    Z+, r17
(0x0006)    st    Z+, r16
(0x0007)    nop
        .dseg
        .org 0x200
        ARR: .byte 5
```

Registers

r16	0xe1
r17	0xe6
⋮	
r30 (ZL)	0x01
r31 (ZH)	0x02
r31:r30 (Z)	0x0201

C **N** **Z** **V**

Data Memory

Address	Contents
0x0200	0xe1
0x0201	0x00
0x0202	0x00
0x0203	0x00
0x0204	0x00
0x0205	0x00
0x0206	0x00
0x0207	0x00
0x0208	0x00
0x0209	0x00

Post-increment instructions can be useful for iterating over arrays, since no extra addition instructions are needed to increment the address.

Arrays (9)

Program Memory

```
(0x0000)    ldi   ZL, low(ARR)
(0x0001)    ldi   ZH, high(ARR)
(0x0002)    ldi   r16, 0xe1
(0x0003)    ldi   r17, 0xe6
(0x0004)    st    Z+, r16
(0x0005)    st    Z+, r17
(0x0006)    st    Z+, r16
(0x0007)    nop
        .dseg
        .org 0x200
        ARR: .byte 5
```

Registers

r16	0xe1
r17	0xe6
⋮	
r30 (ZL)	0x02
r31 (ZH)	0x02
r31:r30 (Z)	0x0202

C **N** **Z** **V**

Data Memory

Address	Contents
0x0200	0xe1
0x0201	0xe6
0x0202	0x00
0x0203	0x00
0x0204	0x00
0x0205	0x00
0x0206	0x00
0x0207	0x00
0x0208	0x00
0x0209	0x00

Post-increment instructions can be useful for iterating over arrays, since no extra addition instructions are needed to increment the address.

Arrays (10)

Program Memory

```
(0x0000)    ldi   ZL, low(ARR)
(0x0001)    ldi   ZH, high(ARR)
(0x0002)    ldi   r16, 0xe1
(0x0003)    ldi   r17, 0xe6
(0x0004)    st    Z+, r16
(0x0005)    st    Z+, r17
(0x0006)    st    Z+, r16
(0x0007)    nop

        .dseg
        .org 0x200
        ARR: .byte 5
```

Registers

r16	0xe1
r17	0xe6
⋮	
r30 (ZL)	0x03
r31 (ZH)	0x02
r31:r30 (Z)	0x0203

C **N** **Z** **V**

Data Memory

Address	Contents
0x0200	0xe1
0x0201	0xe6
0x0202	0xe1
0x0203	0x00
0x0204	0x00
0x0205	0x00
0x0206	0x00
0x0207	0x00
0x0208	0x00
0x0209	0x00

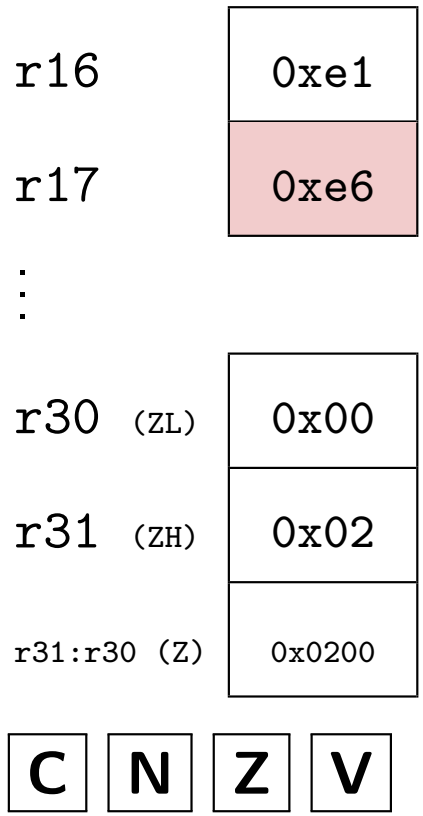
Post-increment instructions can be useful for iterating over arrays, since no extra addition instructions are needed to increment the address.

Arrays (11)

Program Memory

```
(0x0000)    ldi   ZL, low(ARR)
(0x0001)    ldi   ZH, high(ARR)
(0x0002)    ldi   r16, 0xe1
(0x0003)    ldi   r17, 0xe6
(0x0004)    std   Z+1, r16
(0x0005)    std   Z+3, r17
(0x0006)    std   Z+5, r16
(0x0007)    nop
        .dseg
        .org 0x200
        ARR: .byte 5
```

Registers



Data Memory

Address	Contents
0x0200	0x00
0x0201	0x00
0x0202	0x00
0x0203	0x00
0x0204	0x00
0x0205	0x00
0x0206	0x00
0x0207	0x00
0x0208	0x00
0x0209	0x00

Finally, the 'indirect with displacement' addressing mode is provided by LDD and STD. The 'displacement' in this context is a constant offset which is added to the address before the memory access.

Arrays (12)

Program Memory

```
(0x0000)    ldi   ZL, low(ARR)
(0x0001)    ldi   ZH, high(ARR)
(0x0002)    ldi   r16, 0xe1
(0x0003)    ldi   r17, 0xe6
(0x0004)    std   Z+1, r16
(0x0005)    std   Z+3, r17
(0x0006)    std   Z+5, r16
(0x0007)    nop
        .dseg
        .org 0x200
        ARR: .byte 5
```

Registers

r16	0xe1
r17	0xe6
⋮	
r30 (ZL)	0x00
r31 (ZH)	0x02
r31:r30 (Z)	0x0200

C **N** **Z** **V**

Data Memory

Address	Contents
0x0200	0x00
0x0201	0x00
0x0202	0x00
0x0203	0x00
0x0204	0x00
0x0205	0x00
0x0206	0x00
0x0207	0x00
0x0208	0x00
0x0209	0x00

Note that the register pair containing the base address (Z, in this case) is not changed by the instruction.

Arrays (13)

Program Memory

```
(0x0000)    ldi   ZL, low(ARR)
(0x0001)    ldi   ZH, high(ARR)
(0x0002)    ldi   r16, 0xe1
(0x0003)    ldi   r17, 0xe6
(0x0004)    std   Z+1, r16
(0x0005)    std   Z+3, r17
(0x0006)    std   Z+5, r16
(0x0007)    nop
        .dseg
        .org 0x200
        ARR: .byte 5
```

Registers

r16	0xe1
r17	0xe6
⋮	
r30 (ZL)	0x00
r31 (ZH)	0x02
r31:r30 (Z)	0x0200

C **N** **Z** **V**

Data Memory

Address	Contents
0x0200	0x00
0x0201	0xe1
0x0202	0x00
0x0203	0x00
0x0204	0x00
0x0205	0x00
0x0206	0x00
0x0207	0x00
0x0208	0x00
0x0209	0x00

Although the formatting of 'Z+3' looks like assembler-time arithmetic, the displacement constant (1, 3 or 5 in the examples above) is hard-coded into the instruction.

Arrays (14)

Program Memory

```
(0x0000)    ldi   ZL, low(ARR)
(0x0001)    ldi   ZH, high(ARR)
(0x0002)    ldi   r16, 0xe1
(0x0003)    ldi   r17, 0xe6
(0x0004)    std   Z+1, r16
(0x0005)    std   Z+3, r17
(0x0006)    std   Z+5, r16
(0x0007)    nop
        .dseg
        .org 0x200
        ARR: .byte 5
```

Registers

r16	0xe1
r17	0xe6
⋮	
r30 (ZL)	0x00
r31 (ZH)	0x02
r31:r30 (Z)	0x0200

C **N** **Z** **V**

Data Memory

Address	Contents
0x0200	0x00
0x0201	0xe1
0x0202	0x00
0x0203	0x00
0x0204	0x00
0x0205	0x00
0x0206	0x00
0x0207	0x00
0x0208	0x00
0x0209	0x00

Due to bit limitations, only Y and Z can be used with `LDD` and `STD`, and the displacement constant must be a 6-bit unsigned value (in the range $[0, 63]$). Negative displacements are not permitted.

Arrays (15)

Program Memory

```
(0x0000)    ldi   ZL, low(ARR)
(0x0001)    ldi   ZH, high(ARR)
(0x0002)    ldi   r16, 0xe1
(0x0003)    ldi   r17, 0xe6
(0x0004)    std   Z+1, r16
(0x0005)    std   Z+3, r17
(0x0006)    std   Z+5, r16
(0x0007)    nop
        .dseg
        .org 0x200
        ARR: .byte 5
```

Registers

r16	0xe1
r17	0xe6
⋮	
r30 (ZL)	0x00
r31 (ZH)	0x02
r31:r30 (Z)	0x0200

C **N** **Z** **V**

Data Memory

Address	Contents
0x0200	0x00
0x0201	0xe1
0x0202	0x00
0x0203	0x00
0x0204	0x00
0x0205	0x00
0x0206	0x00
0x0207	0x00
0x0208	0x00
0x0209	0x00

(For more thorough examples of arrays and array programming logic, see the posted code examples.)

Program Memory (1)

Instruction	Encoded Instruction	Addr.	Low Byte	High Byte
`lsl r10`		(0x0000)		
`ldi r31, 0`		(0x0001)		
`ldi r31, 0xff`		(0x0002)		
`sts 0x0230,r31`		(0x0003)		
		(0x0004)		
`nop`		(0x0005)		
`jmp 0x0006`		(0x0006)		
		(0x0007)		

Each address in program memory refers to a 16-bit memory location. Addresses are nominally 17-bits, but some parts of the memory subsystem work only with 16-bit program memory addresses (normally allowing only addresses `0x0000` through `0xffff` to be used).

Program Memory (2)

Instruction	Encoded Instruction	Addr.	Low Byte	High Byte
`lsl r10`	`0x0caa`	(0x0000)		
`ldi r31, 0`	`0xe0f0`	(0x0001)		
`ldi r31, 0xff`	`0xefff`	(0x0002)		
`sts 0x0230,r31`	`0x93f00230`	(0x0003)		
		(0x0004)		
`nop`	`0x0000`	(0x0005)		
`jmp 0x0006`	`0x940c0006`	(0x0006)		
		(0x0007)		

The binary encoding of each instruction is either 16 or 32 bits wide. Most instructions have a 16-bit encoding, but instructions which deal with memory (like branches, jumps and load/store instructions) often have 32-bit encodings to allow 16-bit (or larger) memory addresses to be encoded into the instruction.

Program Memory (3)

Instruction	Encoded Instruction	Addr.	Low Byte	High Byte
`lsl r10`	`0x0caa`	(0x0000)		
`ldi r31, 0`	`0xe0f0`	(0x0001)		
`ldi r31, 0xff`	`0xefff`	(0x0002)		
`sts 0x0230,r31`	`0x93f00230`	(0x0003)		
		(0x0004)		
`nop`	`0x0000`	(0x0005)		
`jmp 0x0006`	`0x940c0006`	(0x0006)		
		(0x0007)		

For example, the destination address `0x0230` must be encoded directly into the `STS` instruction above.

Program Memory (4)

Instruction	Encoded Instruction	Addr.	Low Byte	High Byte
`lsl r10`	`0x0caa`	(0x0000)		
`ldi r31, 0`	`0xe0f0`	(0x0001)		
`ldi r31, 0xff`	`0xefff`	(0x0002)		
`sts 0x0230,r31`	`0x93f00230`	(0x0003)		
		(0x0004)		
`nop`	`0x0000`	(0x0005)		
`jmp 0x0006`	`0x940c0006`	(0x0006)		
		(0x0007)		

Since program memory contains 16-bit slots, 32-bit instructions will occupy two consecutive memory locations.

Program Memory (5)

Instruction	Encoded Instruction	Addr.	Low Byte	High Byte
`lsl r10`	`0x0caa`	(0x0000)	`0xaa`	`0x0c`
`ldi r31, 0`	`0xe0f0`	(0x0001)	`0xf0`	`0xe0`
`ldi r31, 0xff`	`0xefff`	(0x0002)	`0xff`	`0xef`
`sts 0x0230,r31`	`0x93f00230`	(0x0003)	`0xf0`	`0x93`
		(0x0004)	`0x30`	`0x02`
`nop`	`0x0000`	(0x0005)	`0x00`	`0x00`
`jmp 0x0006`	`0x940c0006`	(0x0006)	`0x0c`	`0x94`
		(0x0007)	`0x06`	`0x00`

If you inspect program memory on a byte-by-byte level (for example, in an Atmel Studio 'prog FLASH' memory view), you will notice that each 16-bit slot is stored in little-endian order (with the low byte first).

Program Memory (6)

Instruction	Encoded Instruction	Addr.	Low Byte	High Byte
`lsl r10`	`0x0caa`	(0x0000)	`0xaa`	`0x0c`
`ldi r31, 0`	`0xe0f0`	(0x0001)	`0xf0`	`0xe0`
`ldi r31, 0xff`	`0xefff`	(0x0002)	`0xff`	`0xef`
`sts 0x0230,r31`	`0x93f00230`	(0x0003)	`0xf0`	`0x93`
		(0x0004)	`0x30`	`0x02`
`nop`	`0x0000`	(0x0005)	`0x00`	`0x00`
`jmp 0x0006`	`0x940c0006`	(0x0006)	`0x0c`	`0x94`
		(0x0007)	`0x06`	`0x00`

Notice that the 32-bit instructions are not stored in full little-endian order (with all four bytes in reverse order). Instead, the high 16 bits are stored first (in little-endian order), followed by the low 16 bits.

Program Memory (7)

Instruction	Encoded Instruction	Addr.	Low Byte	High Byte
`lsl r10`	`0x0caa`	(0x0000)	`0xaa`	`0x0c`
`ldi r31, 0`	`0xe0f0`	(0x0001)	`0xf0`	`0xe0`
`ldi r31, 0xff`	`0xefff`	(0x0002)	`0xff`	`0xef`
`sts 0x0230,r31`	`0x93f00230`	(0x0003)	`0xf0`	`0x93`
		(0x0004)	`0x30`	`0x02`
`nop`	`0x0000`	(0x0005)	`0x00`	`0x00`
`jmp 0x0006`	`0x940c0006`	(0x0006)	`0x0c`	`0x94`
		(0x0007)	`0x06`	`0x00`

If you are asked to encode or decode instructions on an exam, you should assume that the encoding will be in big-endian order (as in the 'Encoded Instruction' column above) unless otherwise specified.

Program Memory (8)

Instruction	Encoded Instruction	Addr.	Low Byte	High Byte
`lsl r10`	`0x0caa`	(0x0000)	`0xaa`	`0x0c`
`ldi r31, 0`	`0xe0f0`	(0x0001)	`0xf0`	`0xe0`
`ldi r31, 0xff`	`0xefff`	(0x0002)	`0xff`	`0xef`
`sts 0x0230,r31`	`0x93f00230`	(0x0003)	`0xf0`	`0x93`
		(0x0004)	`0x30`	`0x02`
`nop`	`0x0000`	(0x0005)	`0x00`	`0x00`
`jmp 0x0006`	`0x940c0006`	(0x0006)	`0x0c`	`0x94`
		(0x0007)	`0x06`	`0x00`
`.db 0xaa, 0xbb`	`0xbbaa`	(0x0008)	`0xaa`	`0xbb`
`.db -1, -2`	`0xfeff`	(0x0009)	`0xff`	`0xfe`
`.dw 0xcdef`	`0xcdef`	(0x000a)	`0xef`	`0xcd`

Arbitrary data can be stored into program memory using the `.db` directive. Notice that the first byte provided to `.db` becomes the low byte of the 16-bit program memory slot.

Program Memory (9)

Instruction	Encoded Instruction	Addr.	Low Byte	High Byte
`lsl r10`	`0x0caa`	(0x0000)	`0xaa`	`0x0c`
`ldi r31, 0`	`0xe0f0`	(0x0001)	`0xf0`	`0xe0`
`ldi r31, 0xff`	`0xefff`	(0x0002)	`0xff`	`0xef`
`sts 0x0230,r31`	`0x93f00230`	(0x0003)	`0xf0`	`0x93`
		(0x0004)	`0x30`	`0x02`
`nop`	`0x0000`	(0x0005)	`0x00`	`0x00`
`jmp 0x0006`	`0x940c0006`	(0x0006)	`0x0c`	`0x94`
		(0x0007)	`0x06`	`0x00`
`.db 0xaa, 0xbb`	`0xbbaa`	(0x0008)	`0xaa`	`0xbb`
`.db -1, -2`	`0xfeff`	(0x0009)	`0xff`	`0xfe`
`.dw 0xcdef`	`0xcdef`	(0x000a)	`0xef`	`0xcd`

The `.dw` directive stores a word (16-bit value) into program memory in little endian order.

The LPM Instruction (1)

Data memory is **volatile**: Its entire contents are cleared when the power is turned off (or when a soft reset occurs). As a result, it is not possible to initialize arrays in data memory in advance. This can be problematic in cases where a large array of preset values is needed.

Although data memory is cleared after every reset, program memory is not. To maintain arrays of preset values, the values can be planted in program memory by directives like `.db` and `.dw`, then read out of program memory at run time (and copied into data memory, if desired).

The `LPM` (load from program memory) instruction reads a single byte from program memory into a register. Only two encodings are available: `LPM Z, Rd` and `LPM Z+, Rd`.

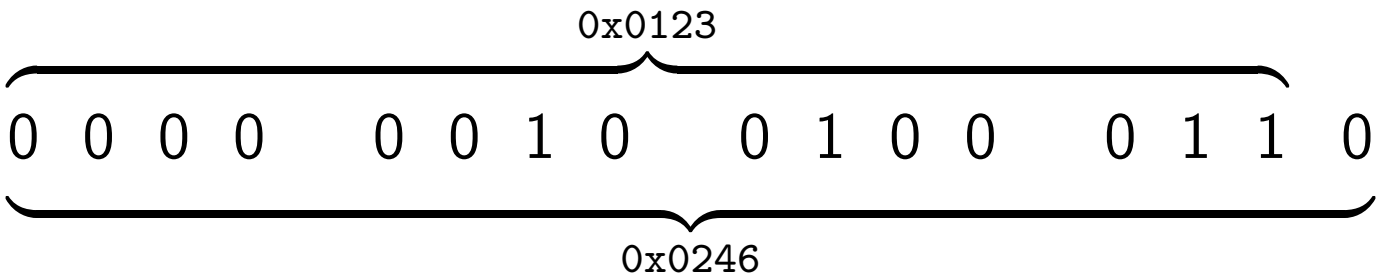
The LPM Instruction (2)

Problem: Suppose we want to use `LPM` to load from program memory at address `0x0123`. Since the result is stored in an 8-bit register, only one byte can be loaded, but address `0x0123` contains two bytes.

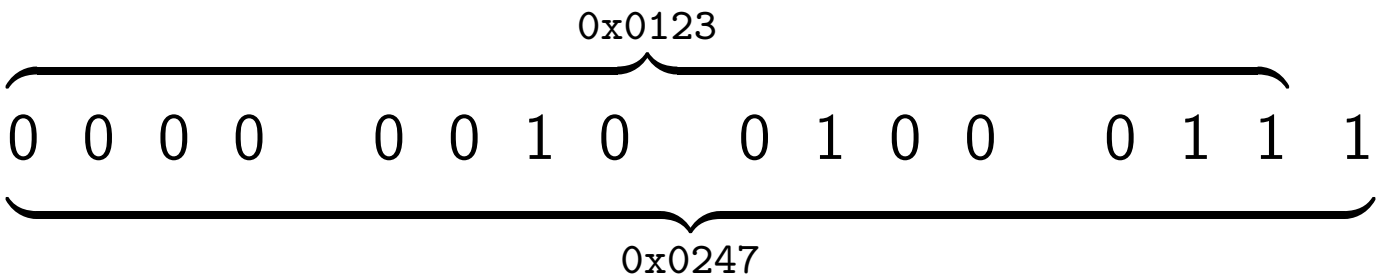
How can we specify which byte to load?

The LPM Instruction (3)

First (low) byte at address `0x0123`:



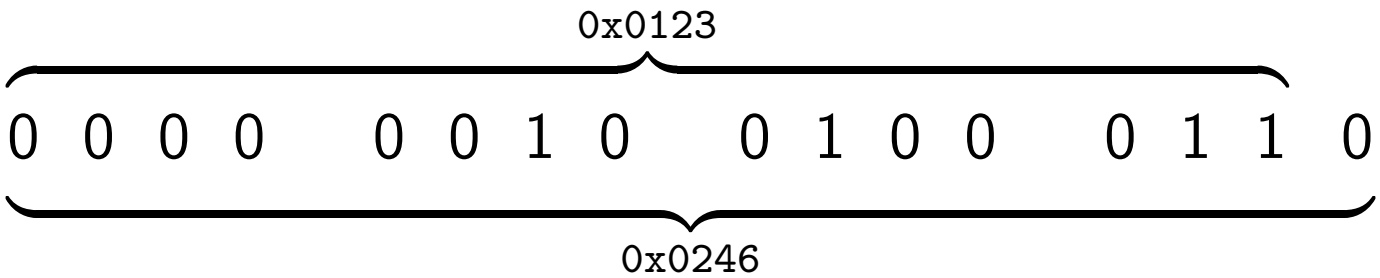
Second (high) byte at address `0x0123`:



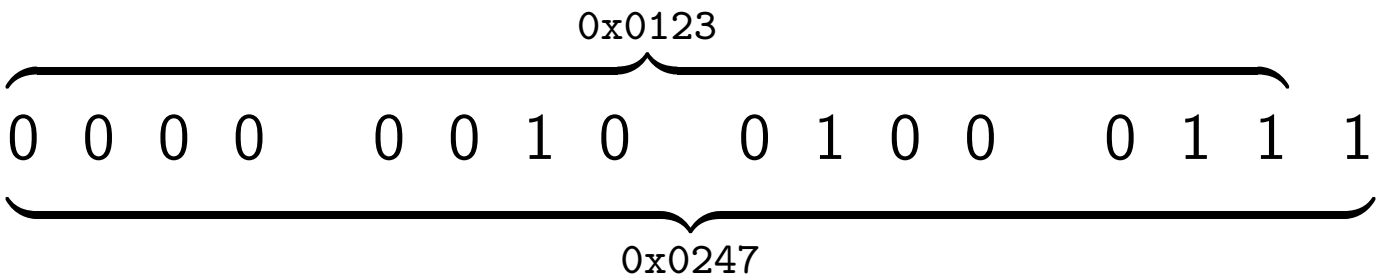
`LPM` uses the lowest bit of the "address" to identify which byte to load and the upper 15 bits to refer to a slot in program memory.

The LPM Instruction (4)

First (low) byte at address 0x0123:



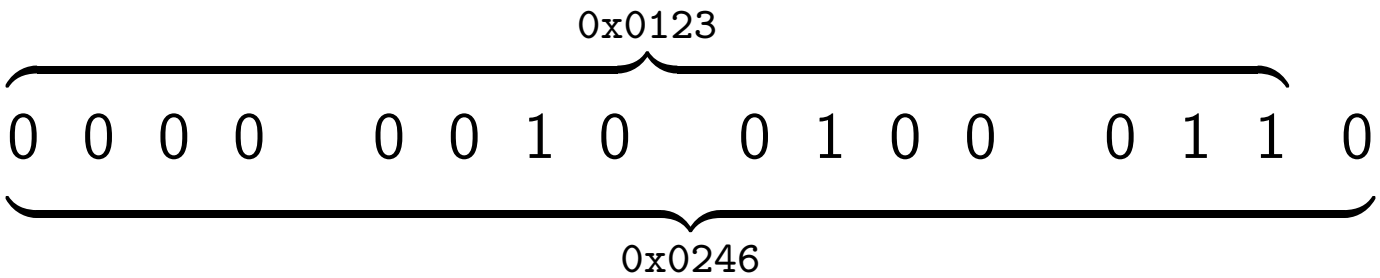
Second (high) byte at address 0x0123:



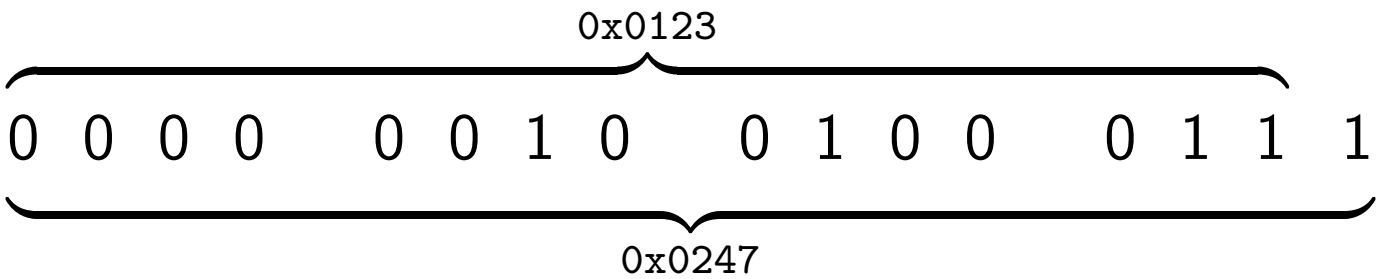
Therefore, to load the first byte at address 0x0123, the address in the Z register would have to be 0x0123 $<<$ 1 $=$ 0x0246 to use `LPM`.

The LPM Instruction (5)

First (low) byte at address 0x0123:



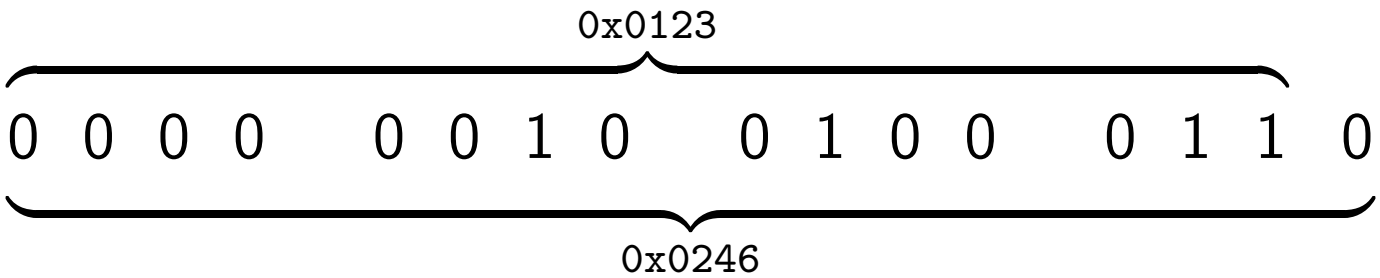
Second (high) byte at address 0x0123:



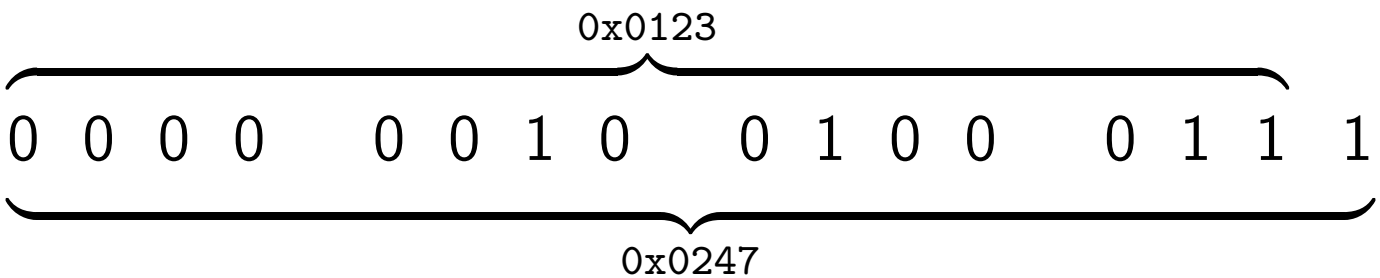
Similarly, to load the second byte at address 0x0123, the address in the Z register would be $(\texttt{0x0123} << 1)|1 = \texttt{0x0247}$.

The LPM Instruction (6)

First (low) byte at address `0x0123`:



Second (high) byte at address `0x0123`:



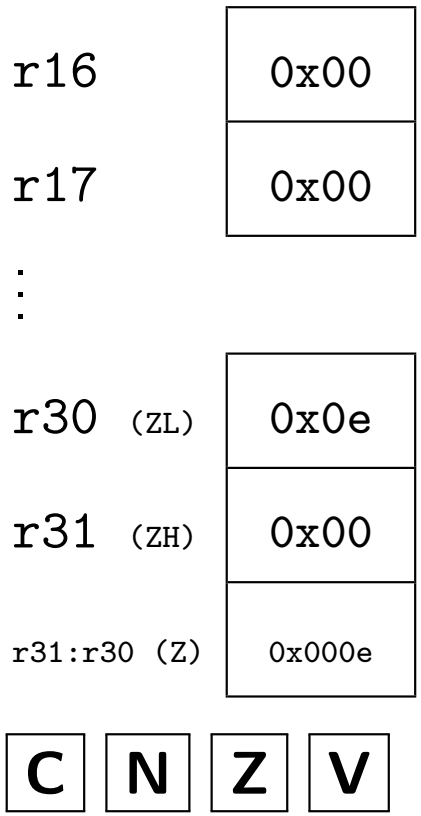
One consequence of this unusual encoding is that `LPM` can only refer to the lowest 2^{15} addresses in program memory (without using extra logic, as described in the instruction set manual page for `LPM`).

The LPM Instruction (7)

Program Memory

```
(0x0000)    ldi   ZL, low(DATA<<1)
(0x0001)    ldi   ZH, high(DATA<<1)
(0x0002)    lpm   r16, Z+
(0x0003)    lpm   r16, Z+
(0x0004)    lpm   r16, Z+
(0x0005)    lpm   r16, Z+
        done:
(0x0006)    rjmp  done
        DATA:
(0x0007)    .db   0xaa, 0xbb
(0x0008)    .db   0x11, 0x12
```

Registers



Data Memory

Address	Contents
0x0200	0x00
0x0201	0x00
0x0202	0x00
0x0203	0x00
0x0204	0x00
0x0205	0x00
0x0206	0x00
0x0207	0x00
0x0208	0x00
0x0209	0x00

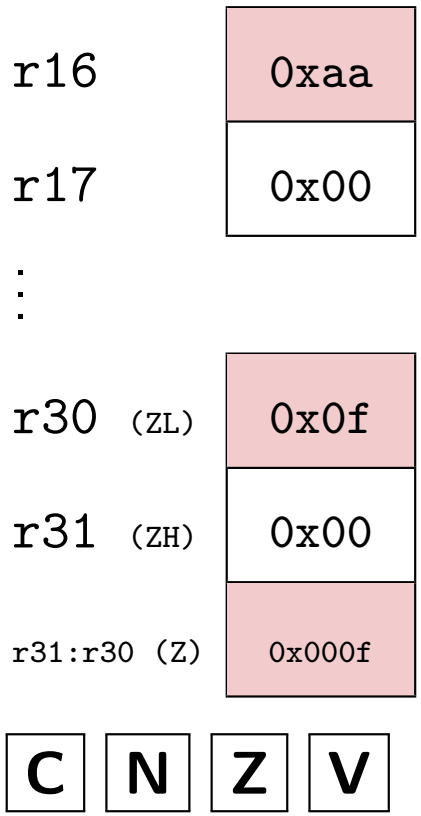
The two `LDI` instructions above load the address of the label `DATA` into the Z register. Notice that the address is shifted **before** splitting it into low and high bytes.

The LPM Instruction (8)

Program Memory

```
(0x0000)    ldi   ZL, low(DATA<<1)
(0x0001)    ldi   ZH, high(DATA<<1)
(0x0002)    lpm   r16, Z+
(0x0003)    lpm   r16, Z+
(0x0004)    lpm   r16, Z+
(0x0005)    lpm   r16, Z+
          done:
(0x0006)    rjmp  done
          DATA:
(0x0007)    .db   0xaa, 0xbb
(0x0008)    .db   0x11, 0x12
```

Registers



Data Memory

Address	Contents
0x0200	0x00
0x0201	0x00
0x0202	0x00
0x0203	0x00
0x0204	0x00
0x0205	0x00
0x0206	0x00
0x0207	0x00
0x0208	0x00
0x0209	0x00

The first LPM instruction loads the low byte at address 0x0007. Notice that after Z is incremented, it contains

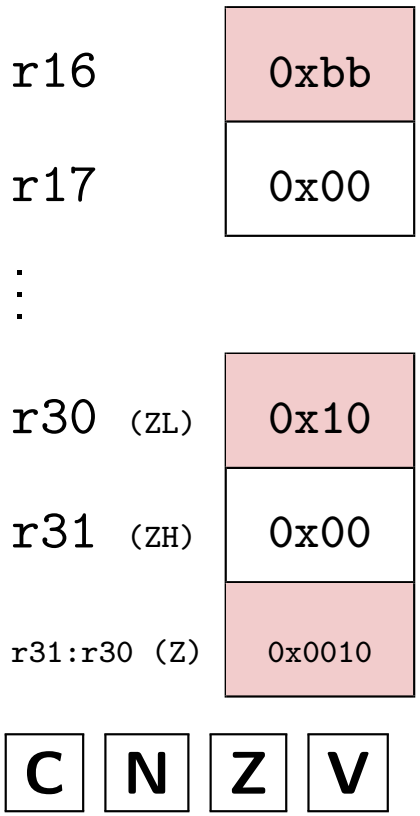
$$\texttt{0x000f} = (\texttt{0x0007} << 1)|1$$

The LPM Instruction (9)

Program Memory

```
(0x0000)    ldi   ZL, low(DATA<<1)
(0x0001)    ldi   ZH, high(DATA<<1)
(0x0002)    lpm   r16, Z+
(0x0003)    lpm   r16, Z+
(0x0004)    lpm   r16, Z+
(0x0005)    lpm   r16, Z+
          done:
(0x0006)    rjmp  done
          DATA:
(0x0007)    .db   0xaa, 0xbb
(0x0008)    .db   0x11, 0x12
```

Registers



Data Memory

Address	Contents
0x0200	0x00
0x0201	0x00
0x0202	0x00
0x0203	0x00
0x0204	0x00
0x0205	0x00
0x0206	0x00
0x0207	0x00
0x0208	0x00
0x0209	0x00

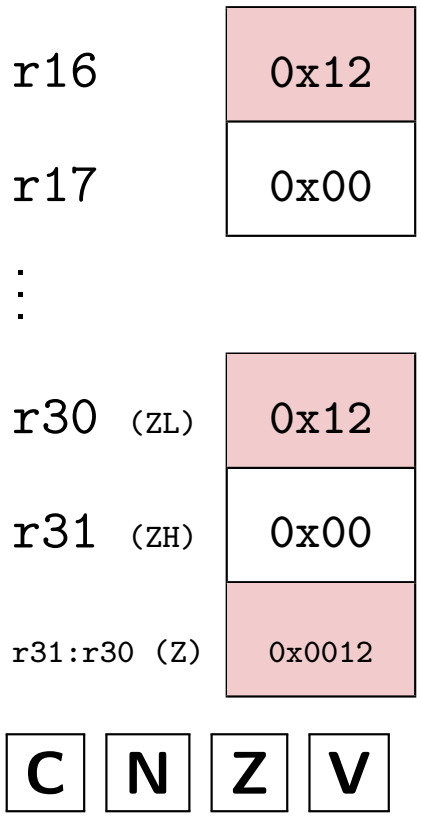
Similarly, after loading from address 0x001c, the incremented value of Z is 0x0010 = 0x0008 << 1.

The LPM Instruction (10)

Program Memory

```
(0x0000)    ldi   ZL, low(DATA<<1)
(0x0001)    ldi   ZH, high(DATA<<1)
(0x0002)    lpm   r16, Z+
(0x0003)    lpm   r16, Z+
(0x0004)    lpm   r16, Z+
(0x0005)    lpm   r16, Z+
        done:
(0x0006)    rjmp  done
        DATA:
(0x0007)    .db   0xaa, 0xbb
(0x0008)    .db   0x11, 0x12
```

Registers



Data Memory

Address	Contents
0x0200	0x00
0x0201	0x00
0x0202	0x00
0x0203	0x00
0x0204	0x00
0x0205	0x00
0x0206	0x00
0x0207	0x00
0x0208	0x00
0x0209	0x00

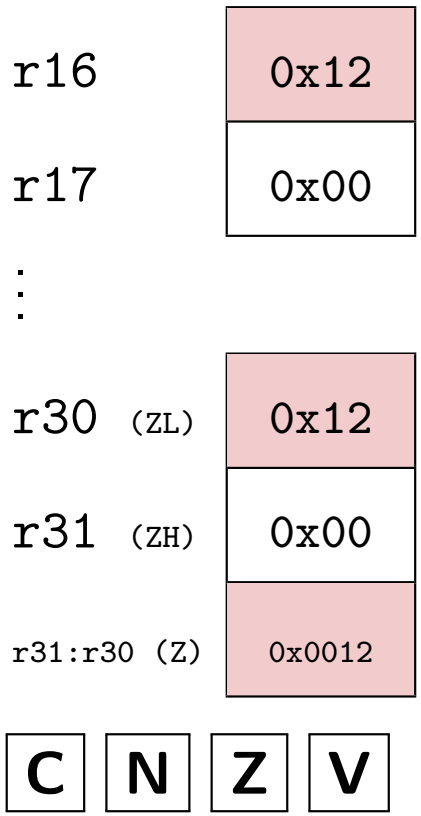
It turns out that a consecutive sequence of bytes placed by `.db` can be read with a consecutive sequence of `LPM` instructions by incrementing Z after each load (similarly to array accesses with `LD` and `ST`).

The LPM Instruction (11)

Program Memory

```
(0x0000)    ldi   ZL, low(DATA<<1)
(0x0001)    ldi   ZH, high(DATA<<1)
(0x0002)    lpm   r16, Z+
(0x0003)    lpm   r16, Z+
(0x0004)    lpm   r16, Z+
(0x0005)    lpm   r16, Z+
        done:
(0x0006)    rjmp  done
        DATA:
(0x0007)    .db   0xaa, 0xbb
(0x0008)    .db   0x11, 0x12
```

Registers



Data Memory

Address	Contents
0x0200	0x00
0x0201	0x00
0x0202	0x00
0x0203	0x00
0x0204	0x00
0x0205	0x00
0x0206	0x00
0x0207	0x00
0x0208	0x00
0x0209	0x00

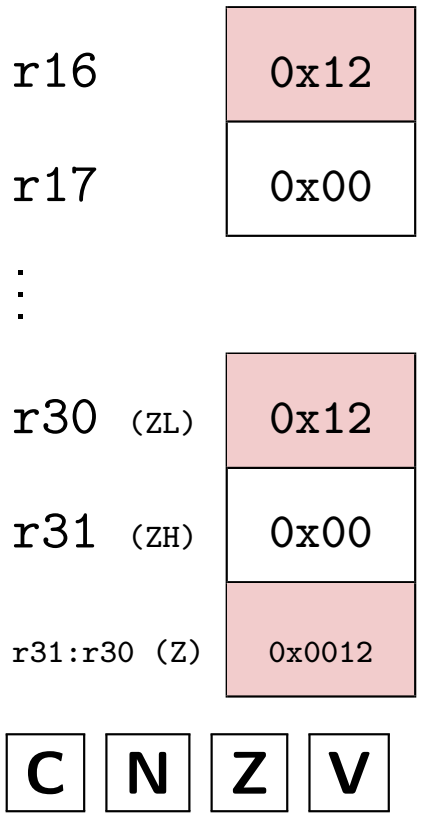
(As with data memory arrays, more thorough examples of the use of LPM can be found in the posted code.)

The LPM Instruction (12)

Program Memory

```
(0x0000)    ldi   ZL, low(DATA<<1)
(0x0001)    ldi   ZH, high(DATA<<1)
(0x0002)    lpm   r16, Z+
(0x0003)    lpm   r16, Z+
(0x0004)    lpm   r16, Z+
(0x0005)    lpm   r16, Z+
        done:
(0x0006)    rjmp  done
        DATA:
(0x0007)    .db   0xaa, 0xbb
(0x0008)    .db   0x11, 0x12
```

Registers



Data Memory

Address	Contents
0x0200	0x00
0x0201	0x00
0x0202	0x00
0x0203	0x00
0x0204	0x00
0x0205	0x00
0x0206	0x00
0x0207	0x00
0x0208	0x00
0x0209	0x00

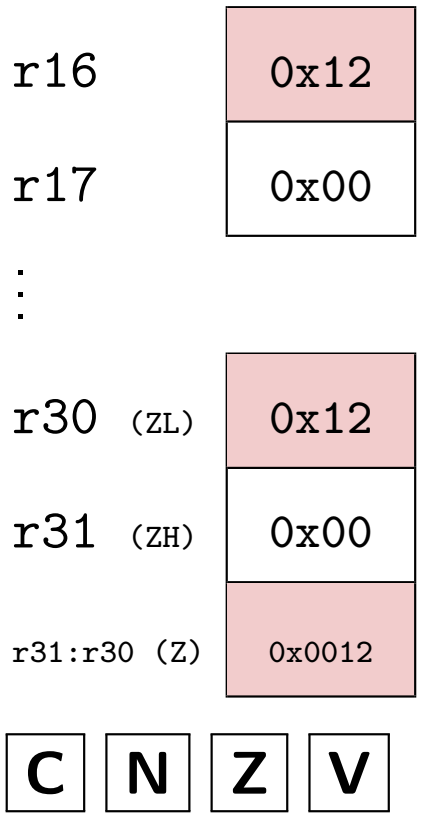
Observation: Since the code starts executing at address 0x0000 and there is an infinite loop at address 0x0006, the program counter will never reach address 0x0007.

The LPM Instruction (13)

Program Memory

```
(0x0000)    ldi   ZL, low(DATA<<1)
(0x0001)    ldi   ZH, high(DATA<<1)
(0x0002)    lpm   r16, Z+
(0x0003)    lpm   r16, Z+
(0x0004)    lpm   r16, Z+
(0x0005)    lpm   r16, Z+
        done:
(0x0006)    rjmp  done
        DATA:
(0x0007)    .db   0xaa, 0xbb
(0x0008)    .db   0x11, 0x12
```

Registers



Data Memory

Address	Contents
0x0200	0x00
0x0201	0x00
0x0202	0x00
0x0203	0x00
0x0204	0x00
0x0205	0x00
0x0206	0x00
0x0207	0x00
0x0208	0x00
0x0209	0x00

Question: What would happen if the `rjmp` instruction were changed to a `nop` instruction?