03- Assembly Language intro

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

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

Assembly Language intro

- Setting the stage
- Microchip AVR Microcontroller
- Machine operation
- Machine language
- AVR assembly language, assembler
- More on the AVR architecture’s ISA (or **Instruction Set Architecture**)

“Lowest possible level for us...”

- The lowest possible level for us is not even low enough for the computer!
  - We need tools to transform our programs and translate them into the lowest level (binary files)
  - Without the binary file for a program, the computer cannot execute the program.
- Brief diversion: **Hello, world!** on a MacBook Pro

Hello, world!

```
$ ./hello
Hello, CSC 230 students!

$ ls -l hello
-rwxr-xr-x  1 zastre  staff  8440 14 Jan 11:06 hello

$ file hello
hello: Mach-O 64-bit executable x86_64
```

Hello, world! (cont.)

```
$ hexdump -C hello
0000000 cf fa ed fe 07 00 00 01 03 00 00 80 02 00 00 00
0000010 0f 00 00 00 b0 04 00 00 85 00 20 00 00 00 00 00
0000020 19 00 00 00 48 00 00 00 5f 5f 50 41 47 45 5a 45
0000030 52 4f 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000040 00 00 00 00 01 00 00 00 00 00 00 00 00 00 00 00
0000050 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000060 00 00 00 00 00 00 00 00 19 00 00 00 d8 01 00 00
0000070 5f 5f 54 45 58 54 00 00 00 00 00 00 00 00 00 00
0000080 00 00 00 00 01 00 00 00 00 10 00 00 00 00 00 00
0000090 00 00 00 00 00 00 00 00 00 10 00 00 00 00 00 00
00000a0 07 00 00 00 05 00 00 00 05 00 00 00 00 00 00 00
00000b0 5f 5f 74 65 78 74 00 00 00 00 00 00 00 00 00 00
...
<snip>
...
00020a0 00 00 00 00 00 00 00 00 02 00 00 00 03 00 00 00
00020b0 00 00 00 40 02 00 00 00 00 00 00 00 5f 5f 6d 68
00020c0 5f 65 78 65 63 75 74 65 5f 68 65 61 64 65 72 00
00020d0 5f 70 72 69 6e 74 66 00 64 79 6c 64 5f 73 74 75
00020e0 62 5f 62 69 6e 64 65 72 00 72 61 64 72 3a 2f 2f
00020f0 35 36 31 34 35 34 32 00
```

Too low-level

- Working directly with binary files is very difficult
  - Only rarely would we ourselves manually craft all of the binary sequences needed for a binary file
  - Modifying such files in a manual way is nearly impossible
- That is why we depend upon so many other tools: they help generate binary files for us
  - compilers, linkers
  - bytecode compilers and virtual machines
  - **assemblers**

Assembler

- A program that translates from human-readable and human–writeable machine code...
  - ... into binary files...
  - ... which are meant for execution on particular computer architecture
- To be able to write a program using an assembler, the programmer needs some knowledge of at least three things:
  1. the computer’s architecture
  2. the computer’s ISA
  3. coding conventions used by assembler

A brief taste of AVR assembly

```
        LDI   R17, 4
        ADD   R16, R17
        DEC   R17
        MOV   R18, R16
END:    RJMP  END
```

1. Need to know architecture has 32 registers, and that the last 16 can be used for all operations {**architecture knowledge**}.
2. Need to know what is meant by LDI, ADD, DEC, MOV and RJMP, and what operands they require {**ISA knowledge**}.
3. Need to know that code-point labels are defined with a colon, and directly used by control-flow instructions {**assembler coding conventions**}.

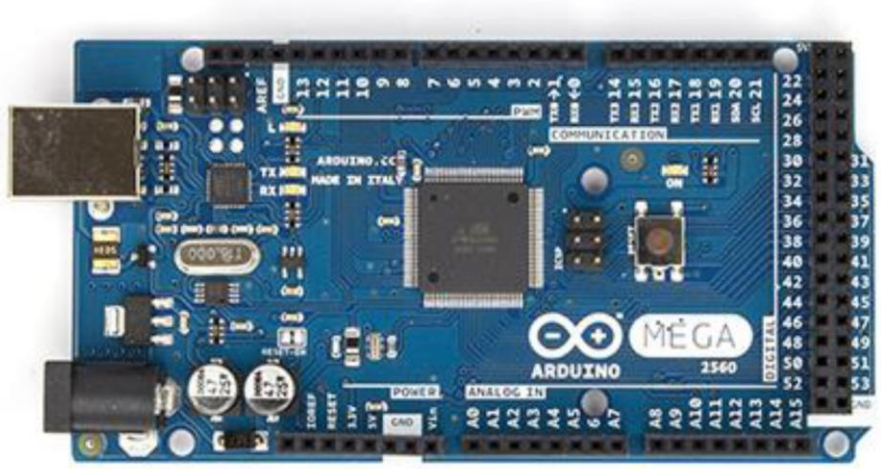
A word to the wise...

- There is a bit of a chicken-and-egg problem here.
  - The three kinds of knowledge are interrelated...
  - ... and so are little bits are learned simultaneously (which can feel confusing).
- Regardless, you have a big advantage!
  - You already know how to express computation in a higher-level language
  - Acquiring that skill required some effort (it isn’t trivial!)
  - And that experience will help here.

Our hardware this term

- Arduino Mega2560 board
  - Arduino is the umbrella label for a number of open-source computer hardware and software projects
  - Also widely available: Arduino Uno and Arduino Nano.
- Our board uses the Microchip **ATmega2560** processor
  - Note: **Microchip** used to be named Atmel
  - Part of the **AVR family of architectures**

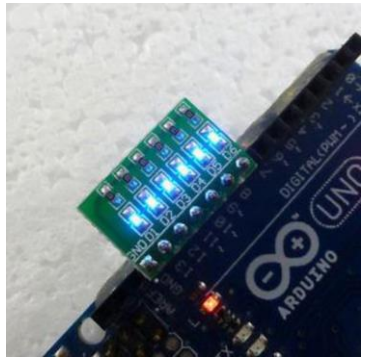
Some lab hardware



Arduino Mega 2560

LCD shield

6-LED strip



More about the Arduino Mega

- Flash memory: 256 kilobytes (KB)
  - 8 KB used by boot loader
  - Your programs will be loaded into flash memory
- Static random-access memory (SRAM): 8KB
  - This will be your main / primary memory
  - Resides on the Mega2560 chip itself
- EEPROM 4KB
  - Will not use it this semester
- Clock speed: 16 MHz
- I/O pins:
  - 54 digital pins
  - 16 analog pins

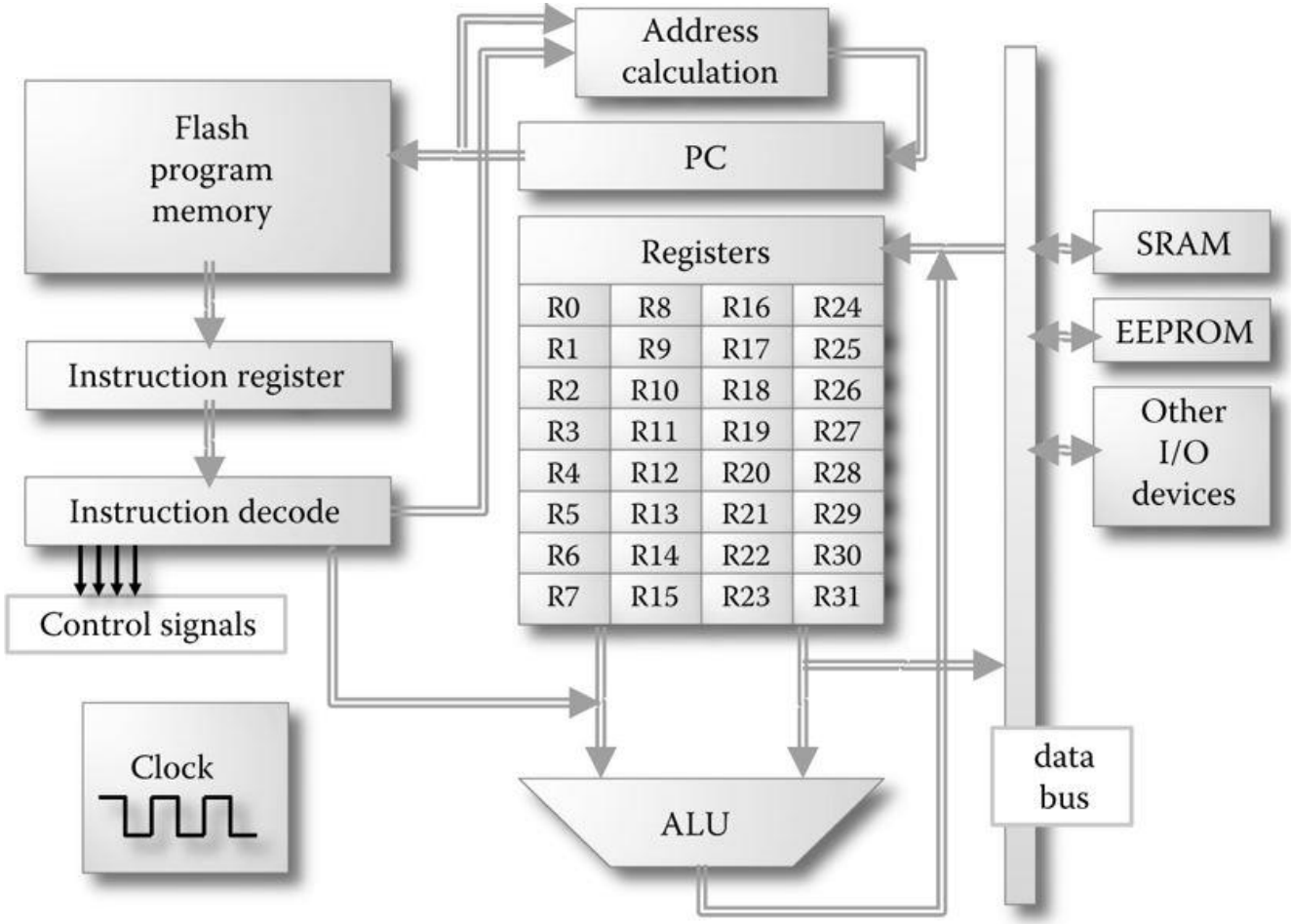
More about the AVR architecture

- AVR is a **Harvard architecture**
- There are a range of AVR processors
  - Arduino Uno and Nano (already mentioned) use the ATMega328P (smaller memory, fewer I/O pins)
  - ATxmega series (larger than ATmega, more I/O pins)
- AVR is a **Reduced Instruction Set Computer** (RISC)
  - Initial architecture conceived in the mid 1990s by two students at the Norwegian Institute of Technology (Alf Egil Bogen, Vegard Wollan)
  - Idea was sold to Atmel (now Microchip)
- AVR processors often also called **microcontrollers**
  - They are indeed computers...
  - ... but intended for use controlling a larger physical device

A bit more about microcontrollers

- Many homes now contain close to 50 microcontrollers
  - digital phones, microwave ovens, dishwashers, clothes washers, television sets, etc.
  - number may get closer to 100 by in the next few years
- Use in current automotive technology
  - Automotive industry instead uses the acronym **ECU** (**electronic control unit**)
  - Average Ford has around 30 such controllers
  - BMW 7-series has up to 150
- Microcontrollers and microprocessors are ever more ubiquitous

Typical AVR core components

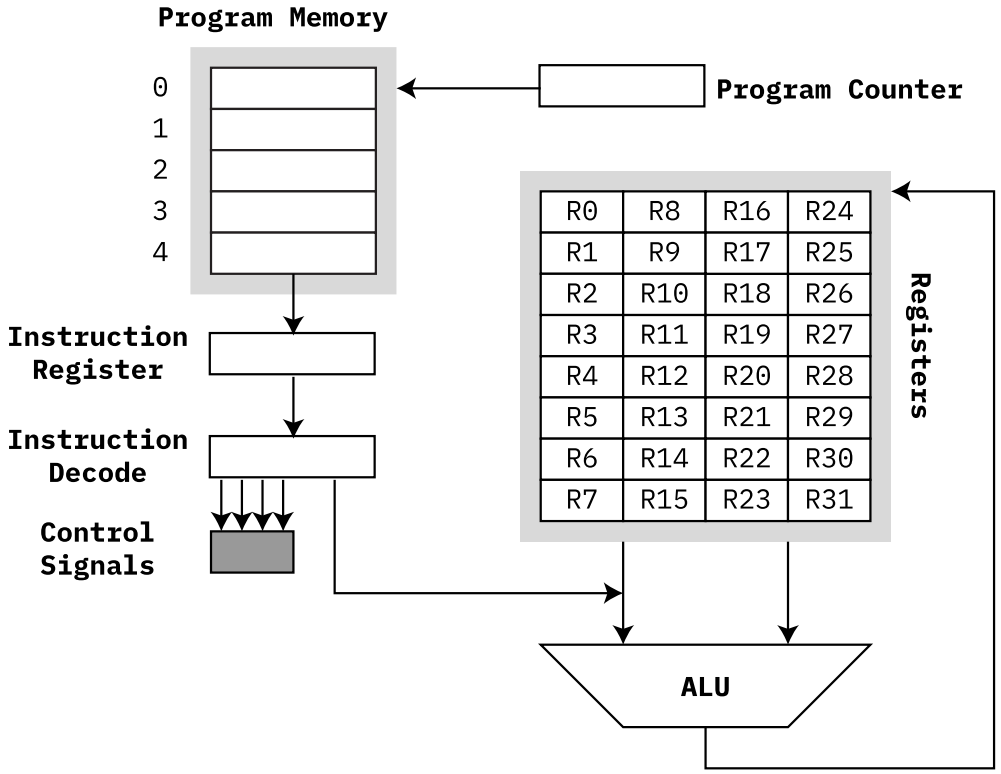


Further AVR features

- Consequences of RISC architecture
  - fixed-length instructions
  - load-store memory access
- Two-stage instruction pipeline
  - First cycle: fetch / decode
  - Second cycle: execution for most instruction types (although a small number of instruction types need extra cycles)
- Wide variety of on-chip peripherals
  - ADCs (analog-to-digital converters)
  - RTC (real-time clock)
  - PWM (pulse width modulator)
  - UART (universal asynchronous reciever/transmitter): for serial communication

1+1=?

- Let’s look at AVR architecture in action
- We will repeatedly use the diagram shown below



1+1=?

```
  LDI  R16, 1
   ADD  R16, R16
LOOP:
   RJMP LOOP
```

Normal opcode convention with two registers:

```
    SOMEOP Rd, Rs
```

means

```
    Rd = Rd <someop> Rs
```

Informally:

Load the immediate value of 1 into register 16.

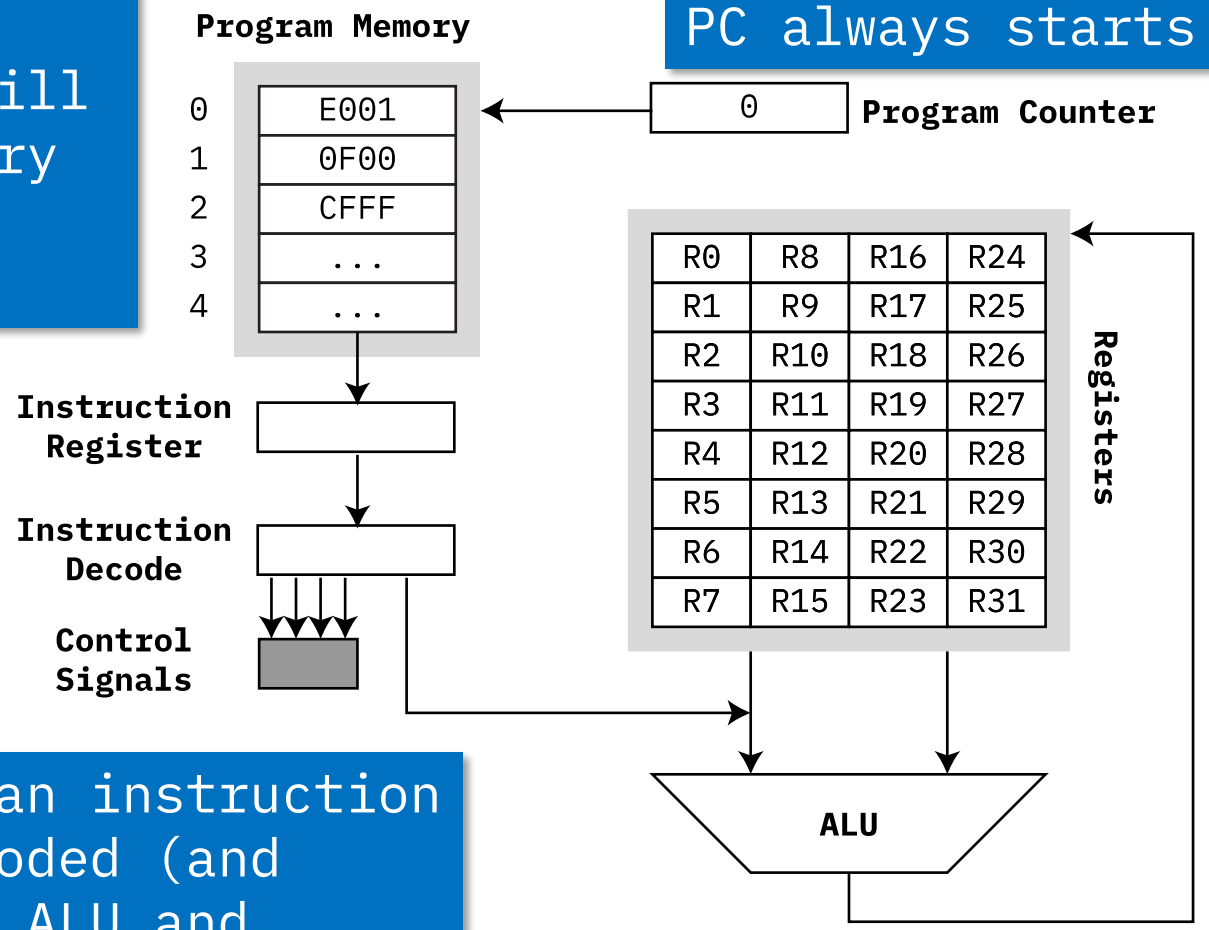
Add the value of R16 to itself, and store the result back into R16.

Finally, go into an infinite loop (i.e., “halt” execution of program)

Before start of first cycle

Program memory contains binary for opcodes. (We will soon learn why these memory values corresponds to the previous slide.)

PC always starts at zero.

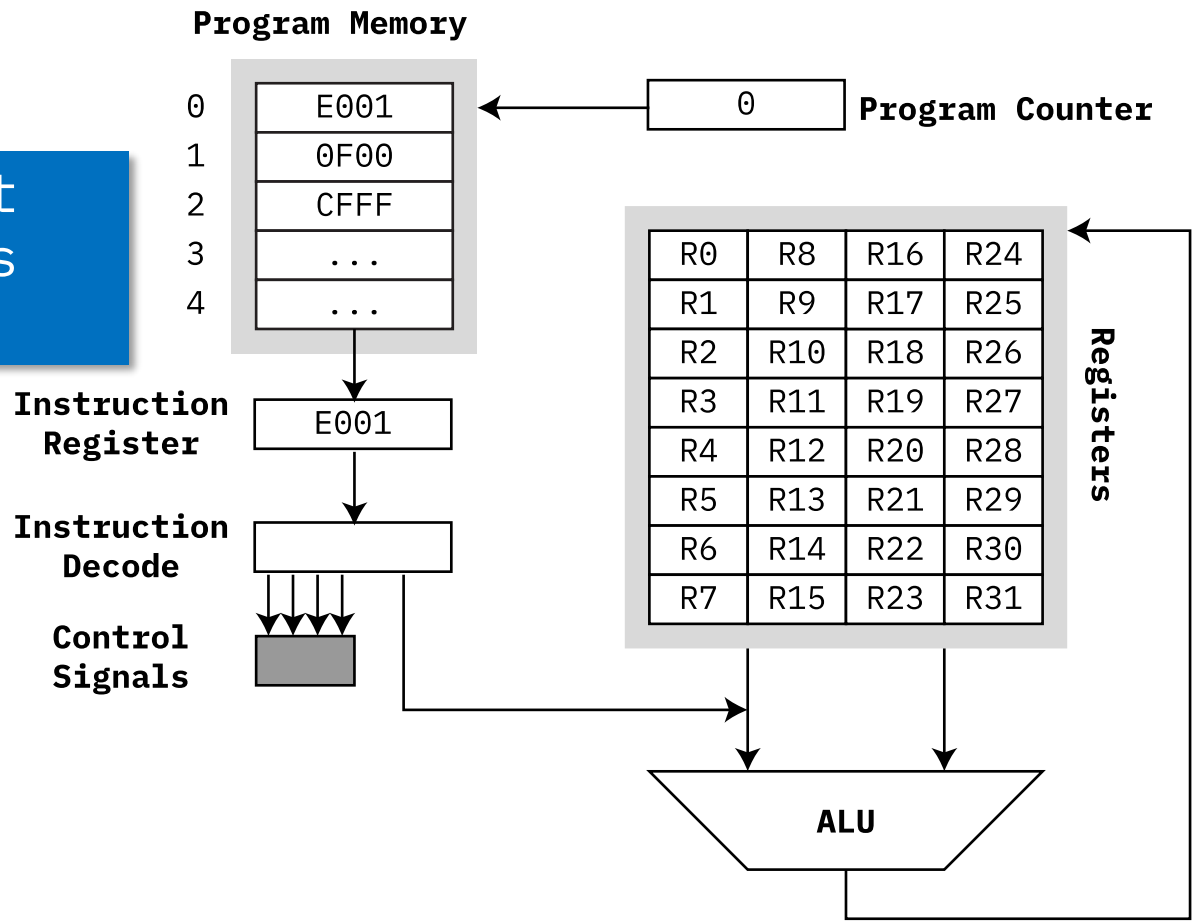


Instruction Register holds encoded instruction while it is being decoded.

Instruction Decode holds an instruction that has already been decoded (and which can now control the ALU and datapaths).

At end of first cycle

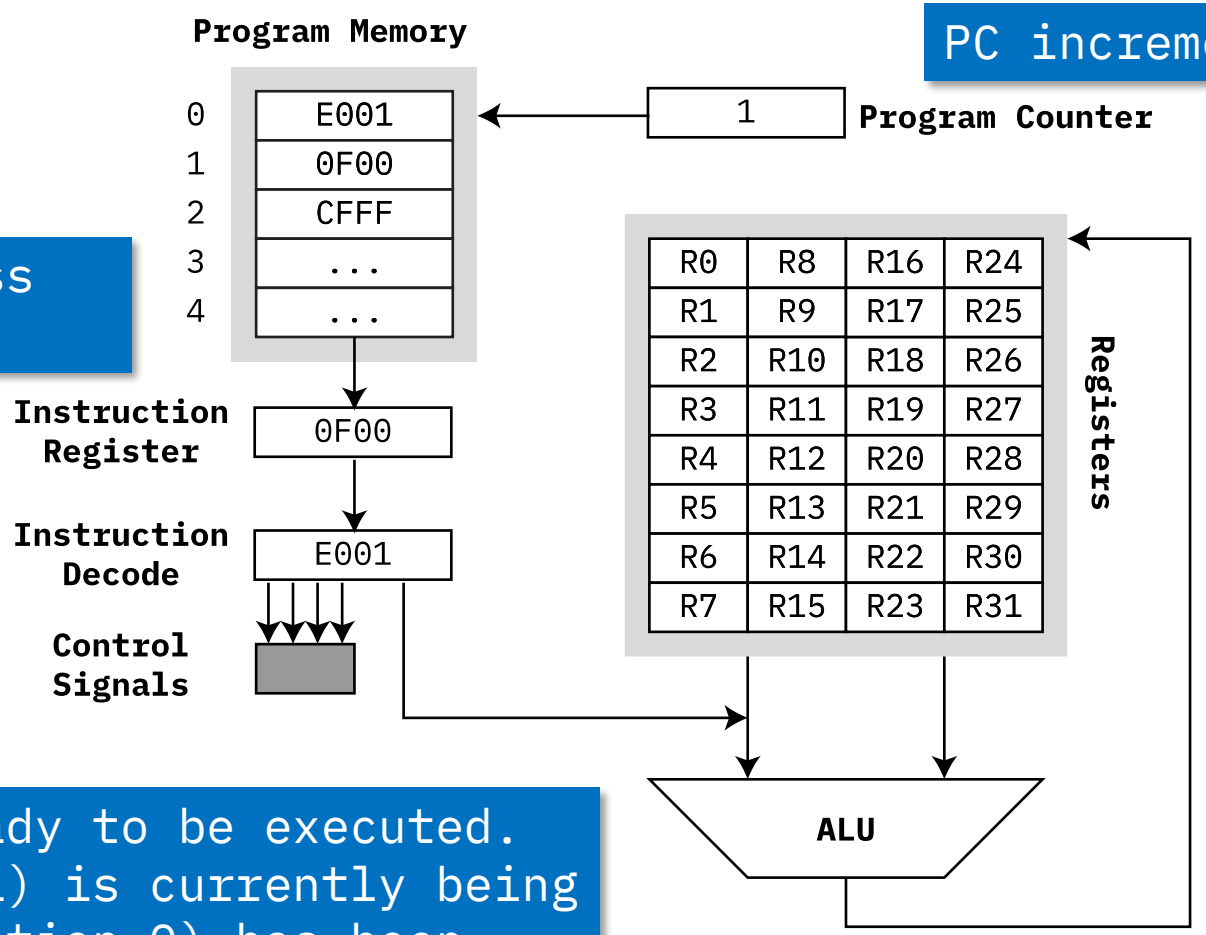
IR contains instruction at address 0, and decoding is nearly finished



At start of second cycle

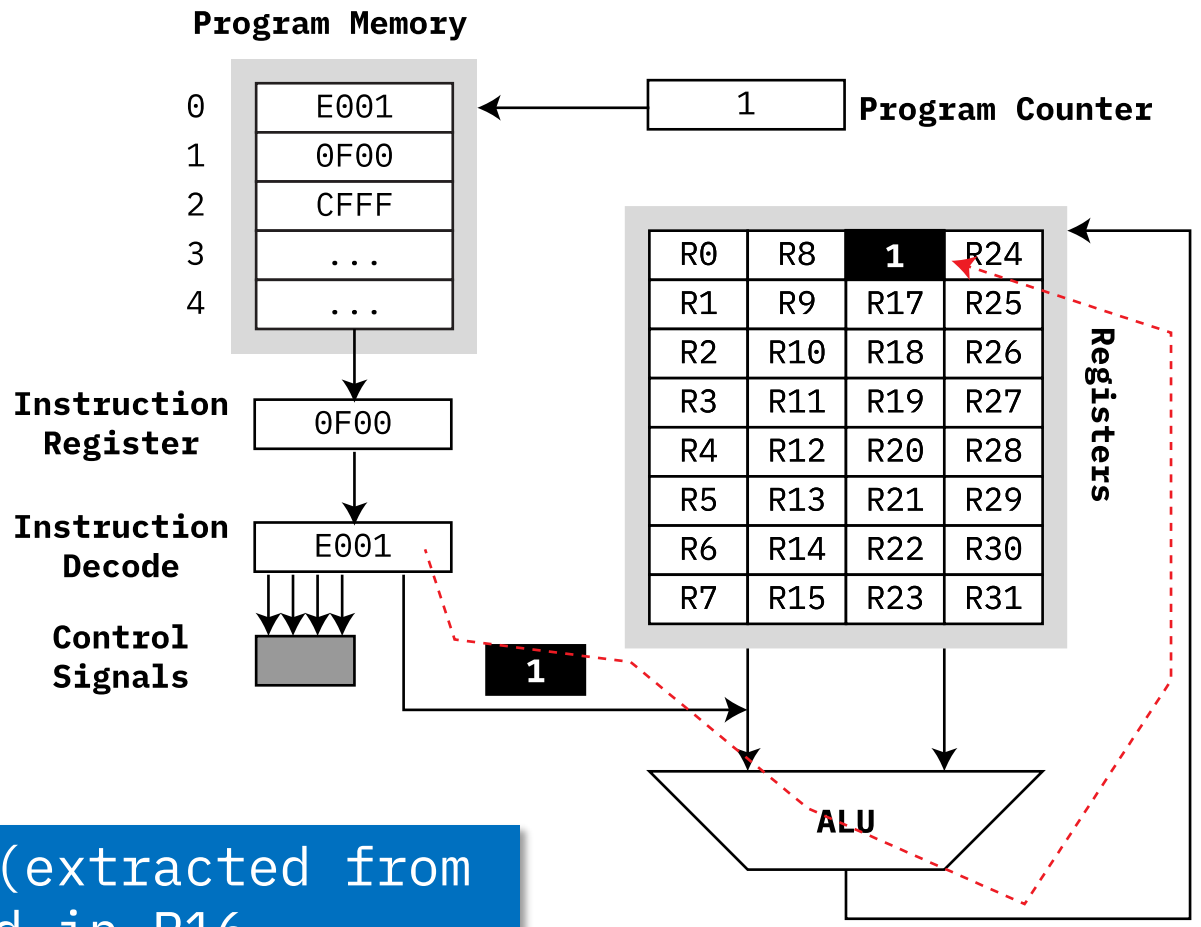
PC incremented by 1

Instruction store add address in PC fetched into IR



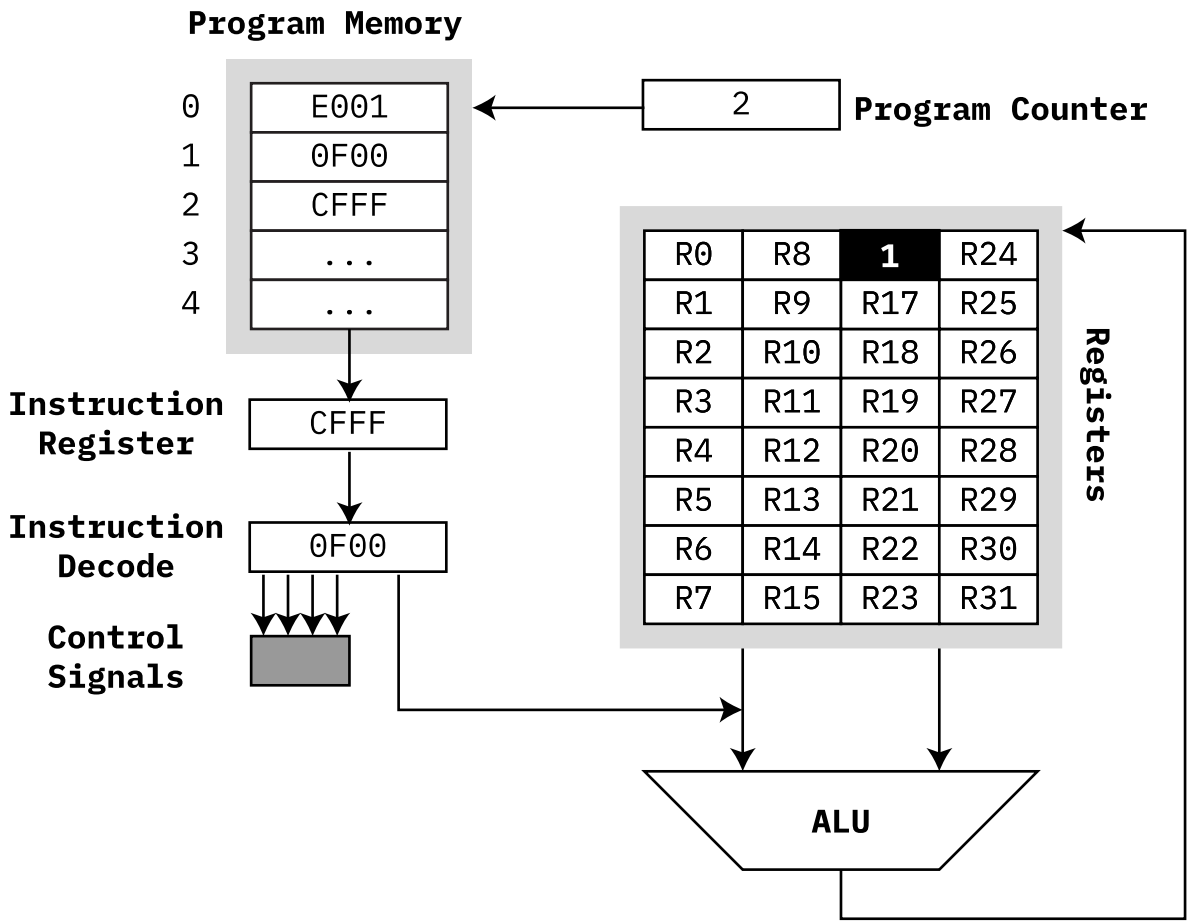
Previous instruction now ready to be executed. That is, 0F00 (instruction 1) is currently being decoded, while E001 (instruction 0) has been decoded and is currently executing.

During (& end) of second cycle

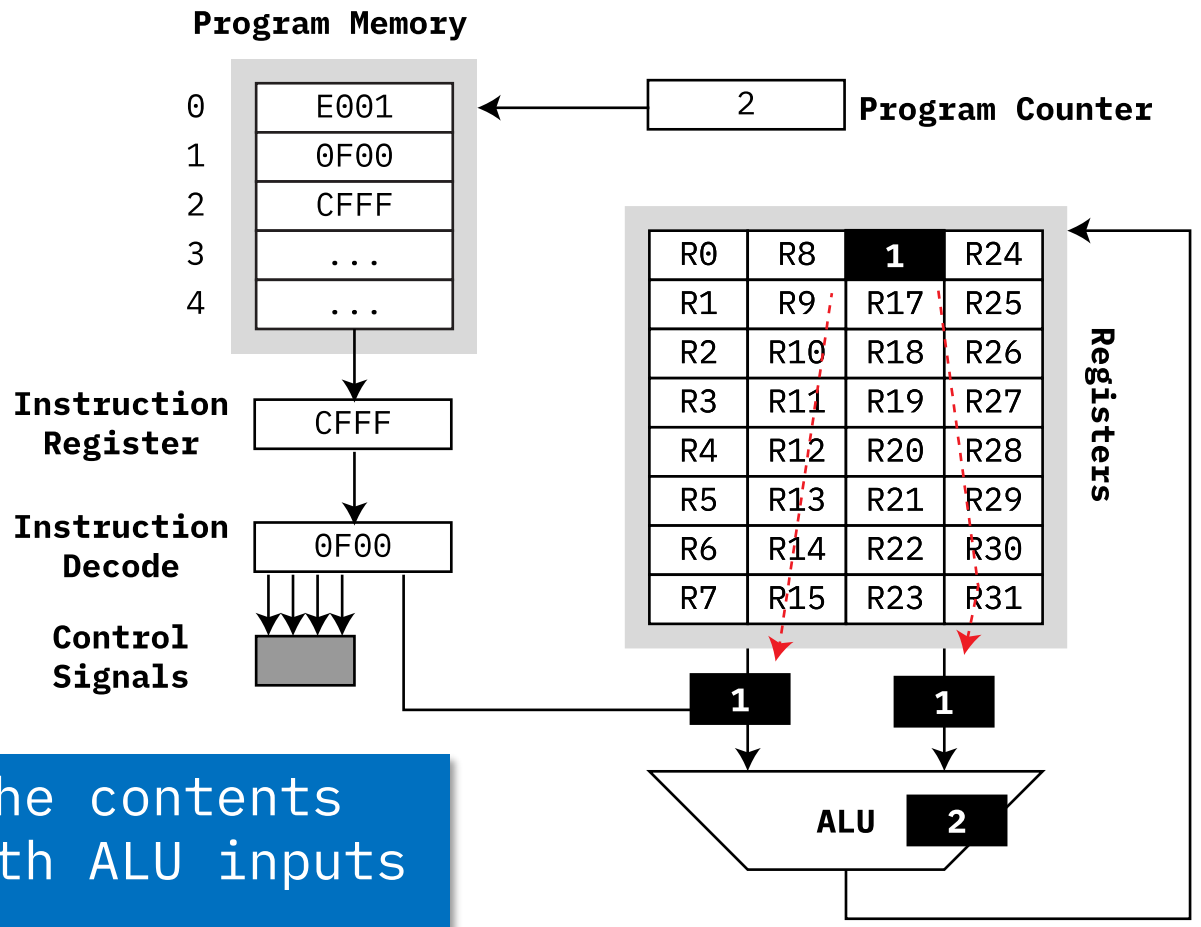


The immediate value of 1 (extracted from the instruction) is stored in R16.

Start of third cycle

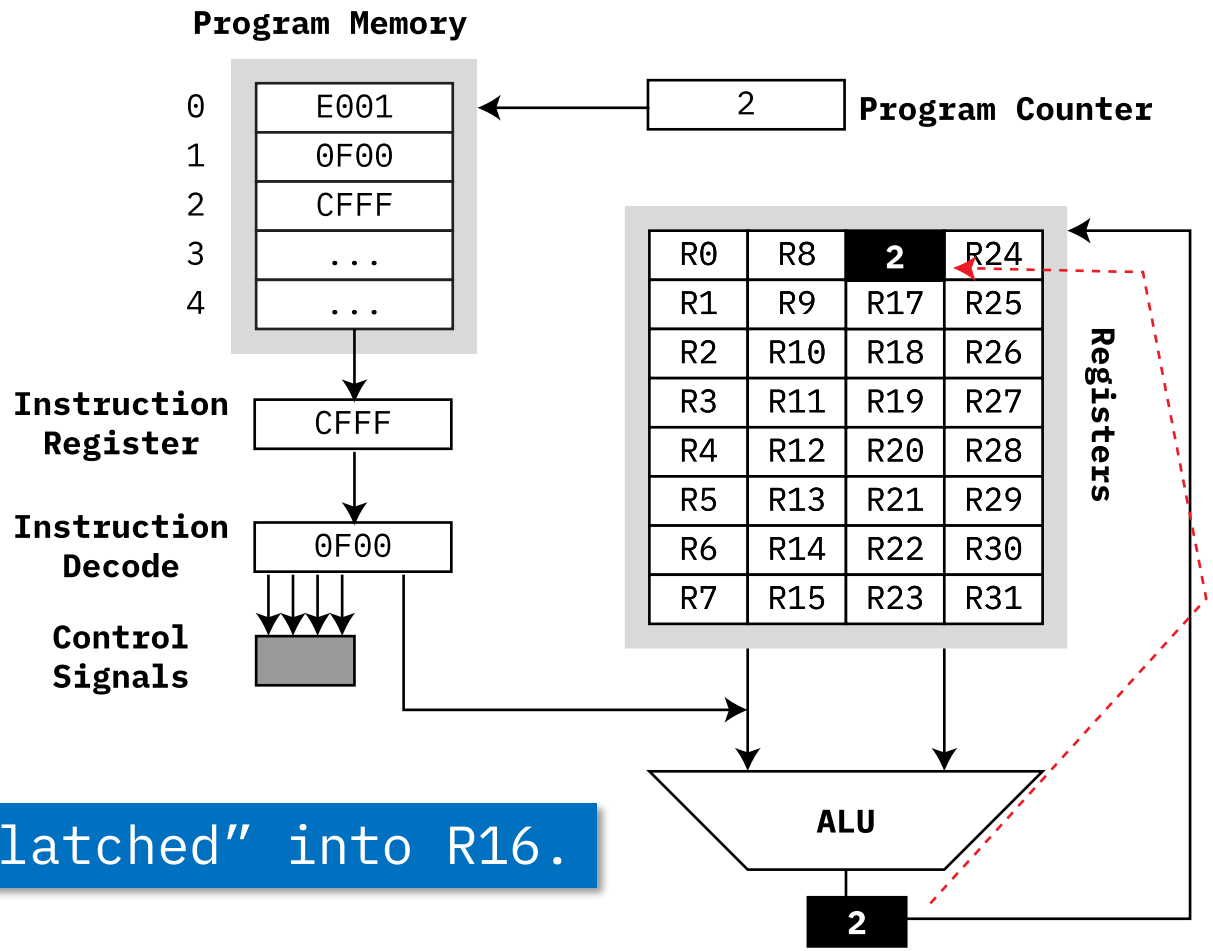


First part of third-cycle execution



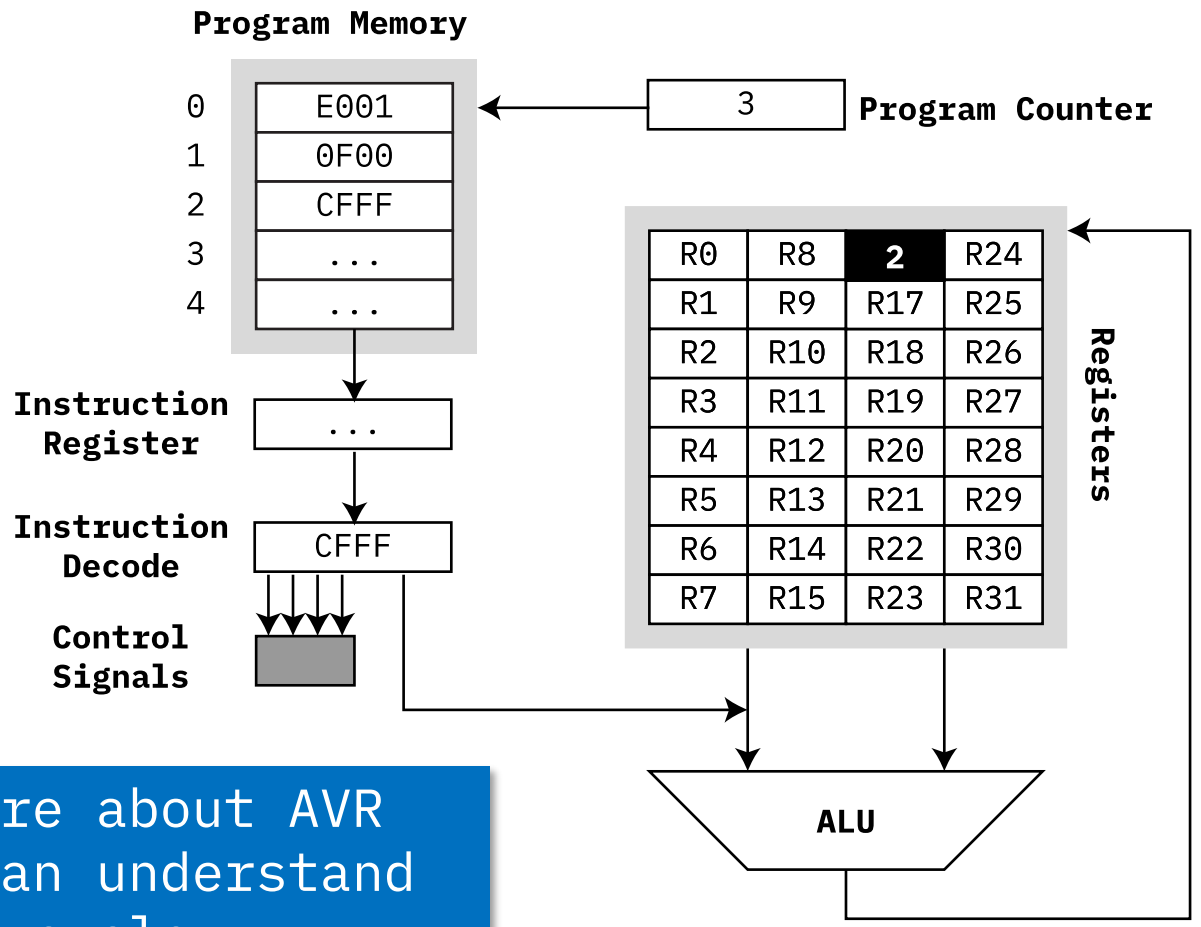
Instruction 0F00 causes the contents of R16 to be placed on both ALU inputs and an ADD performed.

Last part of third-cycle execution



Result of ALU operation “latched” into R16.

Start of fourth cycle



We need to know a bit more about AVR instructions before we can understand what will happen in this cycle.

More about AVR architecture

- Most instructions are 16-bits in length
  - Recall: This is what we call a word (i.e., also two bytes)
  - This therefore means that program memory is treated by the processor as an array of words...
  - ... such as program memory is word addressable.
  - This has some subtle consequences which we'll see
- General purpose registers (R0 to R31)
  - 8-bits in size (i.e., one byte)
  - Data manipulated during any instruction execution cycle must be in one of these registers!
- PC register is, however, 17-bits wide (??!!)

More about AVR architecture (cont.)

- Recall:
  - Instructions are 16-bits wide...
  - ... and are therefore addressed by word.
- Our processor has up to 256 kilo**bytes** of flash memory for programs
  - And this would therefore correspond to 128 kilo**words**.
- PC register must be able to address 217 (i.e., 131072 = 128K) distinct word addresses
  - Therefore our PC register **must be** 17 bits wide
  - Note, though, that some processors in the AVRmega line have less flash memory (128KB & 64KB)
  - Which means some AVRmega processors have a small PC register (16 bits wide & 15 bits wide).

Further into machine language...

- Each machine instruction:
  - has an operation code (or opcode)
  - has zero or more operands
- Operands are:
  - the data, or address of data, to be used in operation
- Opcodes may contain an implicit operand!
- Each machine instruction:
  - occupies some consecutive number of bytes (normally two bytes)
- Each machine program:
  - consists of a sequence of instructions stored in consecutive memory locations
- The Program Counter (PC):
  - contains the address of the next instruction code to be fetched

Move, add, load, store

move: copy between registers	mov Rd, Rr **mov R2, R1**	Copy the contents of R1 into R2 (note order of operands!)
add: add the contents of two registers	add Rd, Rr **add R2, R1**	Add contents of R1 with contents of R2, and store result in R2.
load: loads an 8-bit constant directly to registers 16 to 31	ldi Rd, k **ldi R30, 65**	Load the value 65 into R30. There is a wee gotcha here in that we may only do this with registers R16 to R31.
store: copy contents of a register into a location in memory	st X, Rr **st X, R15**	X is actually two registers R27 & R26, used to create a 16-bit byte address. Therefore the actual location in which R15's content is stored depends upon the values in R27 & R26.

Encoding

- The decoding step is central to a CPU’s operation...
  - ... but what is **encoding**?
- The precise answer depends upon the architecture...
  - ... but the principle idea is seen across different architectures
- Encoding is a step taken by the assembler
- It converts the programmer’s machine code into the equivalent bit sequence for the processor

From the AVR assembly language guide

LDI – Load Immediate

Description:

Loads an 8 bit constant directly to register 16 to 31.

Operation:

(i) Rd ← K

	Syntax:	Operands:	Program Counter:
(i)	LDI Rd,K	$16 \le d \le 31, 0 \le K \le 255$	PC ← PC + 1

16-bit Opcode:

1110	KKKK	dddd	KKKK

From the AVR assembly language guide (cont.)

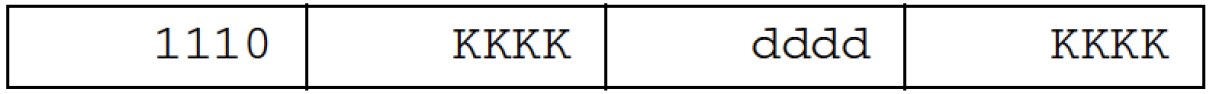
For the LDI instruction

16-bit Opcode:

1110	KKKK	dddd	KKKK

Encoding LDI R21, 151

16-bit Opcode:



Binary for register the register must be offset from 16

21 – 16 = 5 = 0b101 = 0b0101

1110 ???? 0101 ????

Binary for immediate is split into two nibbles.

151 = 0b10010111 high nibble = 1001, low nibble = 0111

1110 1001 0101 0111

0xE957

From the AVR assembly language guide (cont.)

AVR Instruction Set

MOV – Copy Register

Description:

This instruction makes a copy of one register into another. The source register Rr is left unchanged, while the destination register Rd is loaded with a copy of Rr.

Operation:

(i) Rd ← Rr

	Syntax:	Operands:	Program Counter:
(i)	MOV Rd,Rr	$0 \le d \le 31, 0 \le r \le 31$	PC ← PC + 1

16-bit Opcode:

0010	11rd	dddd	rrrr

From the AVR assembly language guide (cont.)

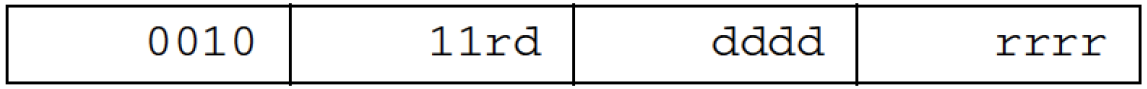
For the MOV instruction

16-bit Opcode:

0010	11rd	dddd	rrrr

Encoding MOV R27, R10

16-bit Opcode:



Binary for **destination** register is a number from 0 to 31

27 = 0b11011

0010 11?1 1011 ????

Binary for source register is also a number from 0 to 31

10 = 0b1010 = 0b01010

0010 1101 1011 1010

0x2DBA

A few observations

- Number of bits used for an opcode may vary between instruction types
  - LDI used four bits
  - MOV used six bits
  - Later we'll see that ADD uses six bits, ST uses 11 bits (!!), etc.
- Sometimes registers are referred to by four bits, sometimes by five
  - Depends upon the register range supported by the instruction
- An operation might also affect more than stated registers
  - Eventually we will do a deep dive on condition codes
  - But for now, a wee taste

From the AVR assembly language guide (cont.)

ADD – Add without Carry

Description:

Adds two registers without the C Flag and places the result in the destination register Rd.

Operation:

(i) Rd ← Rd + Rr

	Syntax:	Operands:	Program Counter:
(i)	ADD Rd,Rr	$0 \le d \le 31, 0 \le r \le 31$	PC ← PC + 1

16-bit Opcode:

0000	11rd	dddd	rrrr

From the AVR assembly language guide (cont.)

Status Register (SREG) and Boolean Formula:

I	T	H	S	V	N	Z	C
–	–	⇔	⇔	⇔	⇔	⇔	⇔

H: $Rd3 \bullet Rr3 + Rr3 \bullet \overline{R3} + \overline{R3} \bullet Rd3$
Set if there was a carry from bit 3; cleared otherwise

S: $N \oplus V$, For signed tests.

V: $Rd7 \bullet Rr7 \bullet \overline{R7} + \overline{Rd7} \bullet \overline{Rr7} \bullet R7$
Set if two's complement overflow resulted from the operation; cleared otherwise.

N: R7
Set if MSB of the result is set; cleared otherwise.

Z: $\overline{R7} \bullet \overline{R6} \bullet \overline{R5} \bullet \overline{R4} \bullet \overline{R3} \bullet \overline{R2} \bullet \overline{R1} \bullet \overline{R0}$
Set if the result is $00; cleared otherwise.

C: $Rd7 \bullet Rr7 + Rr7 \bullet \overline{R7} + \overline{R7} \bullet Rd7$
Set if there was carry from the MSB of the result; cleared otherwise.

R (Result) equals Rd after the operation.

Jump

- Recall our original four-line program
- Normally:
  - When a program is executed in an OS environment...
  - ... the program returns control back to the OS
- However, what if we have no OS in the environment?
  - This will be the situation with our Arduino board

```
  LDI  R16, 1
  ADD  R16, R16
LOOP:
  RJMP LOOP
```

Program termination

- We can create an infinite loop to avoid processing (possibly) garbage instructions in the programming memory following our last useful instruction
- Idiomatic way of doing this in AVR is by using the relative jump instructions
- In essence, the instruction modifies the address used for a fetch...
- ... and overwrites the already-incremented PC with a different address

From the AVR assembly language guide (cont.)

RJMP – Relative Jump

Description:

Relative jump to an address within PC - 2K +1 and PC + 2K (words). For AVR microcontrollers with Program memory not exceeding 4K words (8K bytes) this instruction can address the entire memory from every address location. See also JMP.

Operation:

(i) PC ← PC + k + 1

	Syntax:	Operands:	Program Counter:	Stack
(i)	RJMP k	-2K ≤ k < 2K	PC ← PC + k + 1	Unchanged

16-bit Opcode:

1100	kkkk	kkkk	kkkk

RJMP

- Relative Jump
  - “relative” means in relation to the current PC address
  - the argument in the opcode is a value k
  - action of instruction is: PC = PC + 1 + k
- In our infinite loop, we want to jump back our RJMP instruction itself
  - That is, we want the effect of the operation to be that of making it look as if the PC is unchanged from instruction to instruction
- Subtle point:
  - In the AVR document, references to the PC are always to the instruction’s location in memory
- Therefore we use k = -1
  - k must therefore be a two’s complement number
  - But how many bits?

Encoding RJMP LOOP

	Syntax:	Operands:
(i)	RJMP k	$-2K \leq k < 2K$

16-bit Opcode:

1100	kkkk	kkkk	kkkk

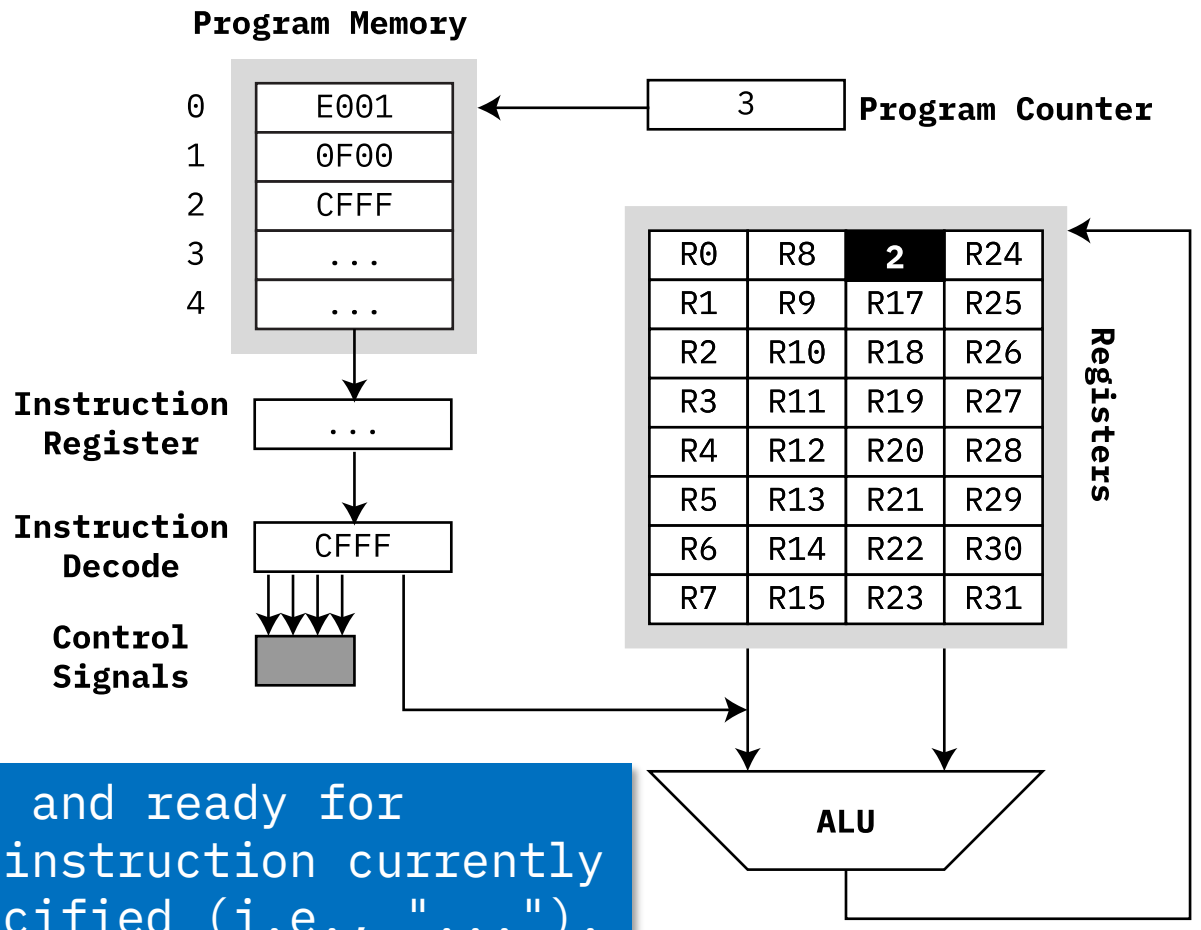
k is a 12-bit number (i.e., values from -2048 up to **but not including** 2048)

-1 = 0b111111111111

1100 1111 1111 1111

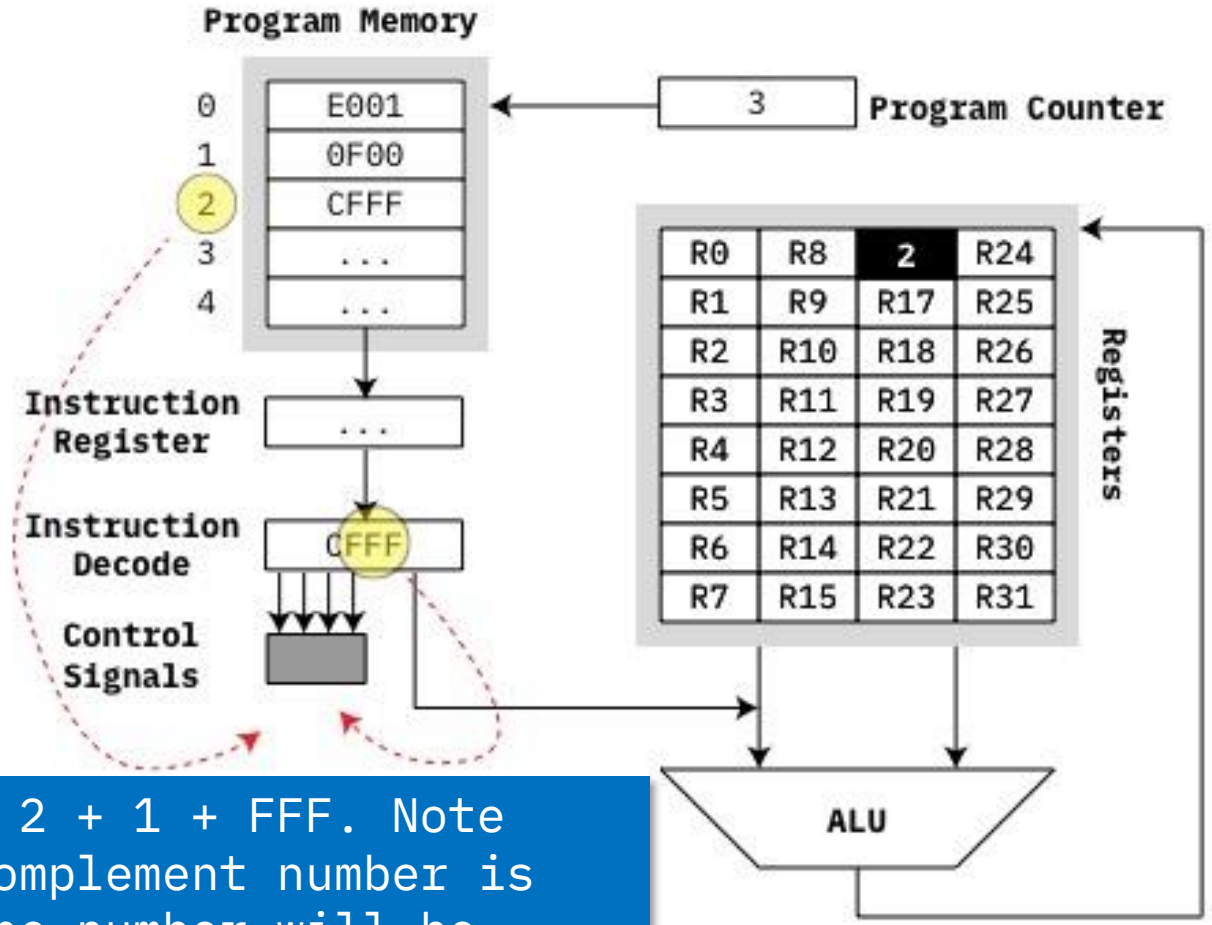
0xCFFF

Return to the start of four cycle



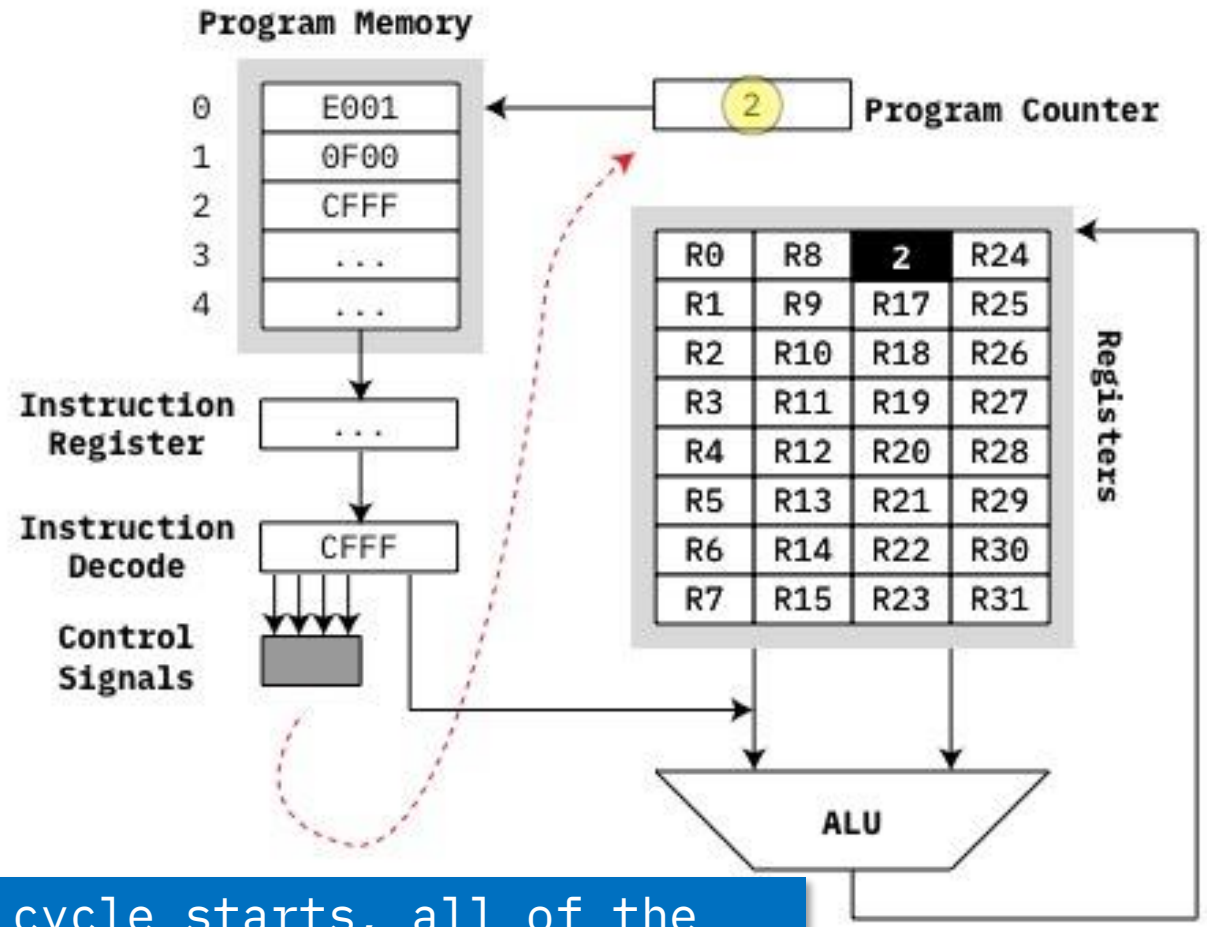
Instruction that was decoded, and ready for execution, is RJMP LOOP. The instruction currently undergoing encoding isn't specified (i.e., "..."). It doesn't matter for our example.

In the midst of the fourth cycle



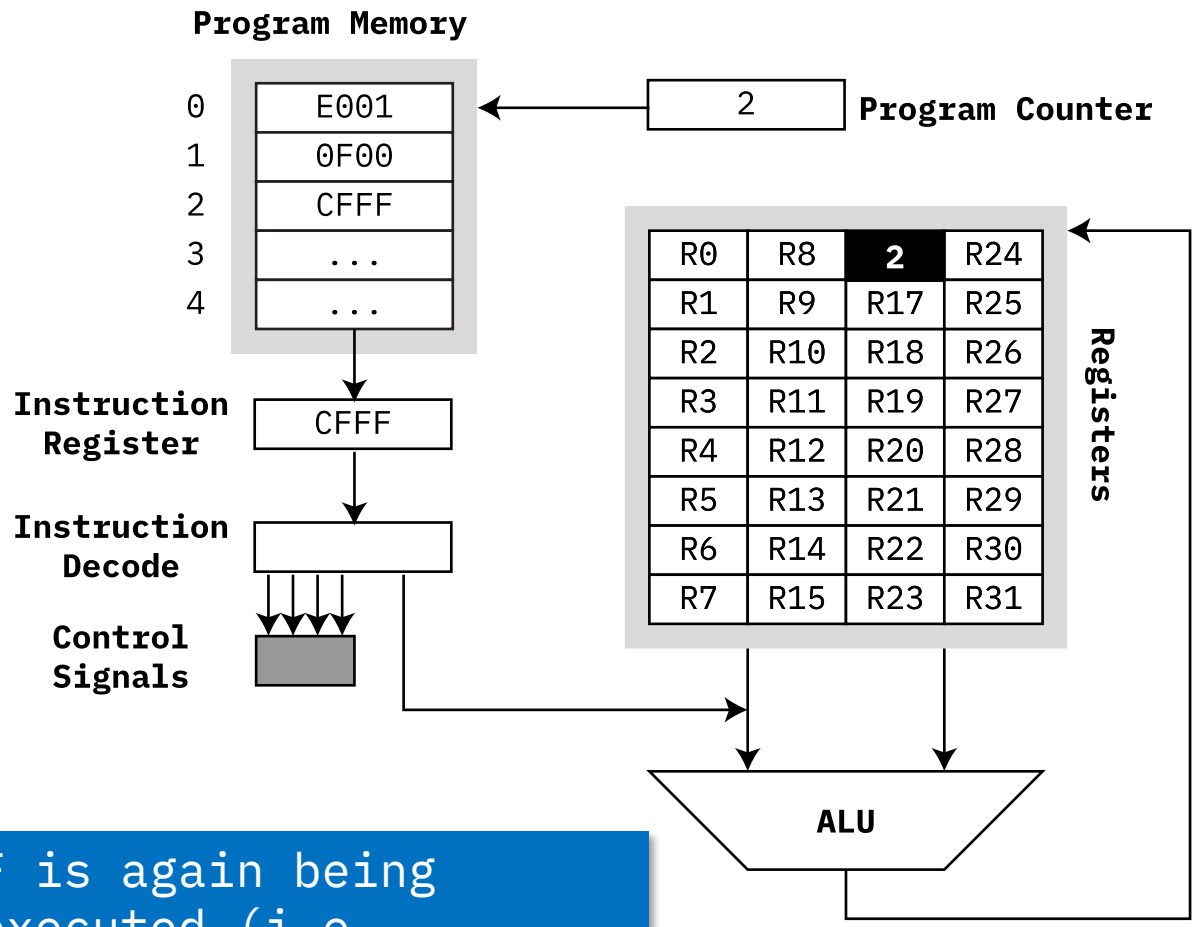
The control unit calculates: 2 + 1 + FFF. Note that when the 12-bit two's complement number is used in 16-bit arithmetic, the number will be signed extended (i.e., FFF becomes FFFF).

Later on at end of fourth cycle



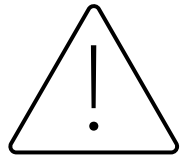
PC is updated. When the next cycle starts, all of the work done to decode "..." will be discarded (i.e., we'll have one cycle where no execution occurs).

Start of fifth cycle



A cycle now begins where CFFF is again being decoded, but now nothing is executed (i.e., similar to the very first cycle shown earlier).

Potential gotcha (first warning)



- In our example the PC had values of 0, 1, 2 and 3
  - Yet instructions are two bytes (i.e., 16 bits) wide, However, we normally refer to memory locations by byte.
  - What is happening here?
- Remember!
  - AVR architecture refers to program instructions by word address ...
  - ... and to user data by byte address
- How can we convert a byte address into a word address?
  - And what problems/issues might result if we are not careful?

Looping again

- The only loop we have seen so far is an unconditional loop
  - This infinite loop solves a problem involving program termination
  - Different systems solve the problem in other ways
- Suppose, however, we want a loop that has a condition?
  - Sort of like a for-loop

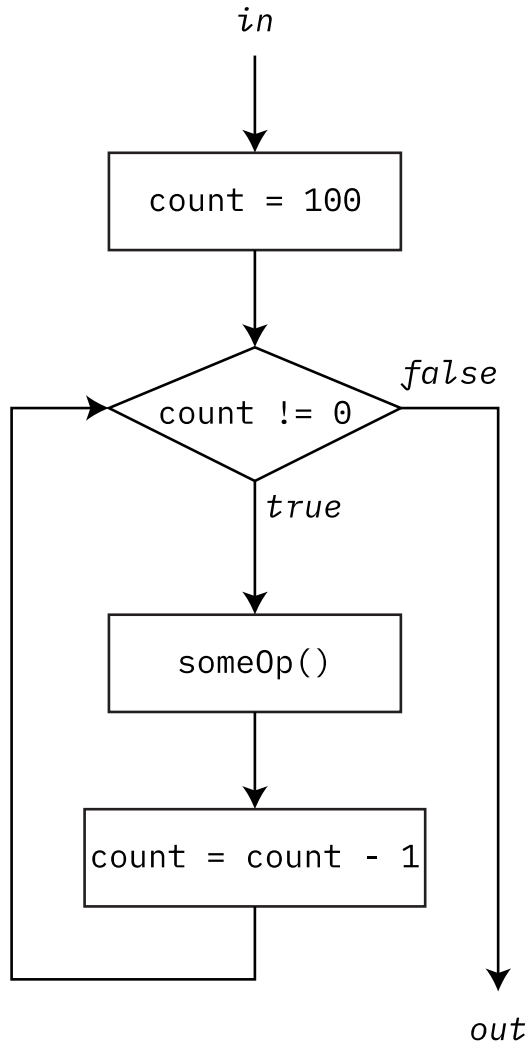
Loop example (Java syntax)

```
for (int counter = 100; counter > 0; counter--) {
    someOperation();
}
```

```
// Equivalent...

for (int counter = 100; counter != 0; counter--) {
    someOperation();
}
```

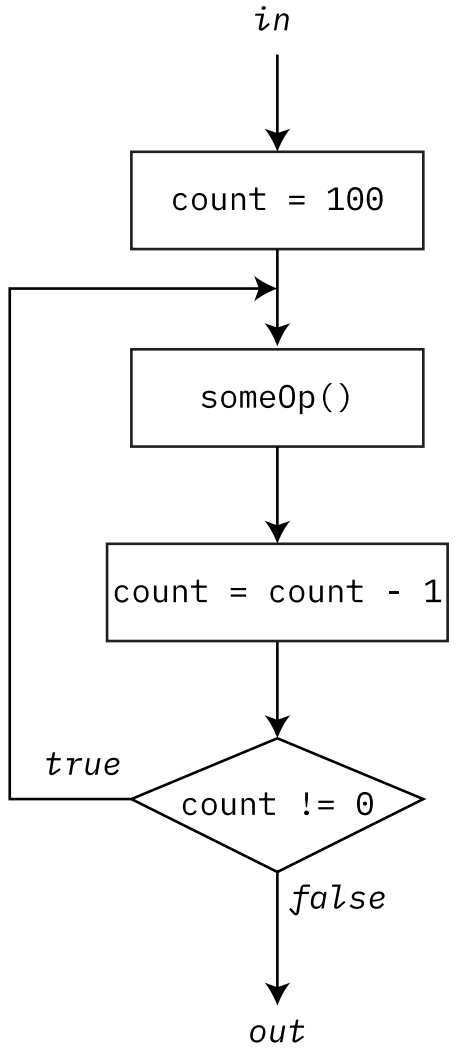
Loop example (Java syntax) (cont.)

```
// Equivalent...

for (int count = 100; count != 0; count = count - 1)
{
    someOp();
}
```



Loop example (Java syntax) (cont.)

```
// Equivalent...

int count = 100;
do {
    someOp();
    count = count - 1;
} while (count != 0);
```



AVR Assembly

- When thinking about conditional branching in assembly...
- ...it is best to think in terms of flowcharts!
  - This is because far more control-flow possibilities exist in assembly than in Java...
  - ... and so “thinking in Java” could be misleading.
- We must also separate out the action to test a condition from the branching action itself
  - And sometimes the action is not even needed!
- Example: AVR loop to perform an operation 10 times
  - To keep it simple, the operation will be a NOP
- But we need one more piece...

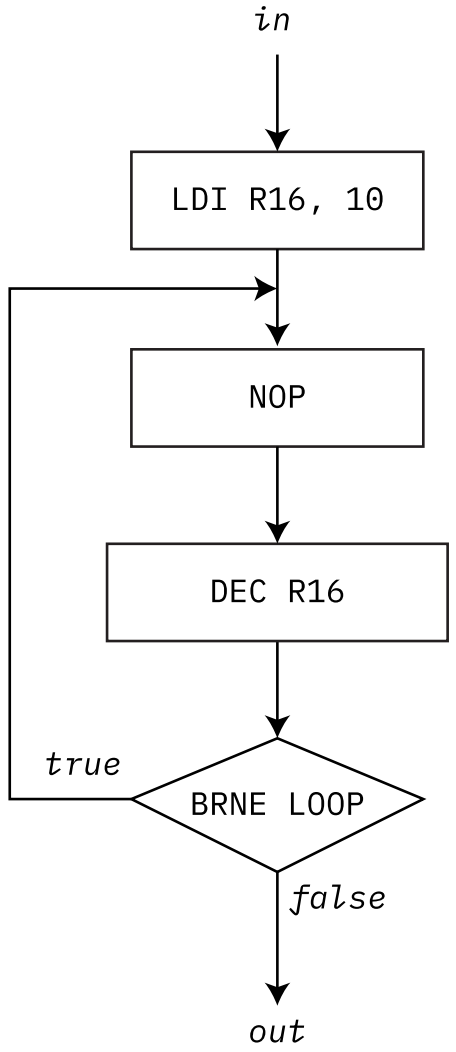
AVR Status Register

- AVR architecture maintains an 8-bit register that indicates the status after operations
- We'll look at most of the bits (or, more appropriately, "flags") in this course...
- ... but for now the most important is the **Zero Flag**
- If the result of the most recent arithmetic operation is the value zero, **the Zero Flag is set to true**
- When two values are compared and are equal in value, **the Zero Flag is set to true**

A loop in assembly

```
        LDI R16, 0x0A   ; A
LOOP:   NOP             ; B
        DEC R16         ; C
        BRNE LOOP       ; D

END:    RJMP END        ; E
```

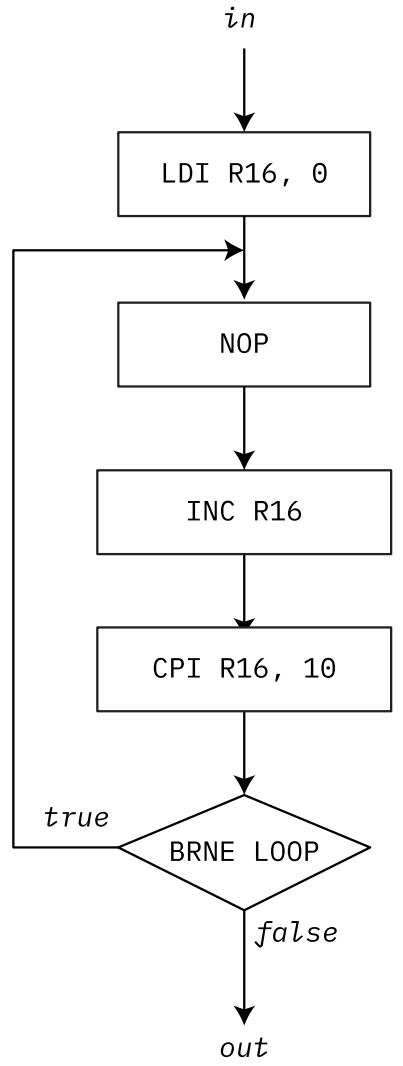
BRNE causes the branch to occur if the Zero Flag is set to false (if the flag is not “cleared”)



Similar loop (but counting up instead)

```
       LDI R16, 0x00   ; A
LOOP:  NOP             ; B
       INC R16         ; C
       CPI R16, 0x0A   ; D
       BRNE LOOP       ; E

END:   RJMP END        ; F
```

The CPI instruction will ensure the Zero Flag is set or cleared as appropriate given the comparison (i.e., if operands are the same, zero flag is set, otherwise it is cleared).



Temporary summary on branches

- There are a large number of branch opcodes (around 20)
- Perhaps the two most useful are:
  - BREQ (branch on equal to zero)
  - BRNE (branch on not equal to zero)
  - These check if the Z flag of the status register is set or cleared
  - Opcodes such as CP and CPI set (and clear) the Z flag
- Some of the other branch opcodes will be described later in the course
  - When we look arithmetic using AVR instructions

Reading & writing from memory

- Recall that the AVR architecture is Harvard architecture
  - The program memory is in one memory space...
  - ... and the user’s data in is another memory space
- Therefore when accessing memory other than registers:
  - ... we must indicate the address ...
  - **... and use the correct instruction** for the memory space.
  - (We’ll pretty much ignore EEPROM in this course.)

Reading and writing user data (SRAM)

- Direct load and store
  - LDS, STS
  - <addr> is 16 bits
  - <addr> is also usually defined with the help of the assembler.
- Indirect load and store
  - The value stored in X can be either static (i.e., known at assembly time)...
  - ... or dynamic (i.e., computed at runtime)
- **What is the *X* of which we speak?**

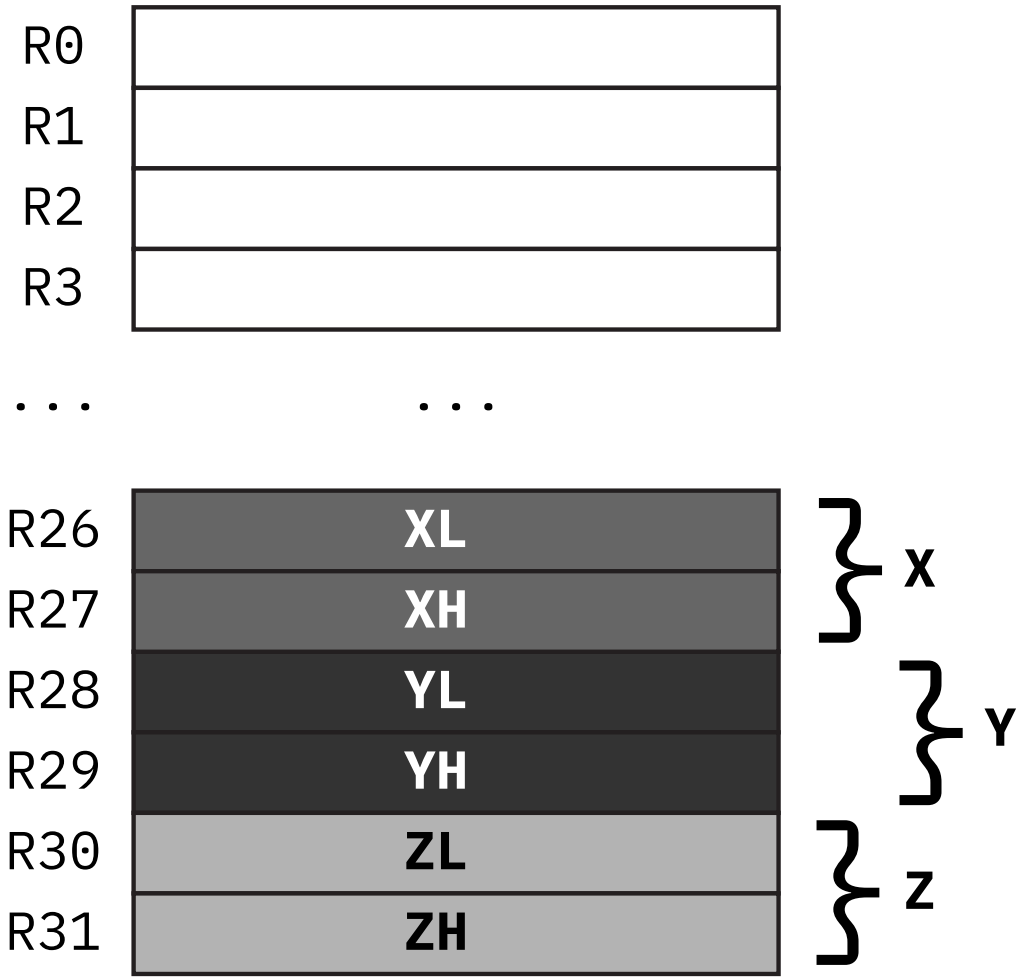
```
; examples

LDS R3, <addr>

STS <addr>, R2
```

```
; examples

LD R3, X

ST X, R2
```

Pseudo registers

- A possible source of confusion is that some architectural elements may go by more than one name
- Example: registers X, Y and Z
  - Used for reads and writes from SRAM
  - For working with memory we need 16 bits
  - Therefore: X is the concatenation of bits in R27 followed by bits in R26
  - That is, X=R27:R26 (where “:” means “bit concatenation”)
  - Similar for Y and Z (R29:R28 and R31:R30)
- X, Y and Z are also called **pointers**

Register File & Pointer Structure



Pseudo registers (cont.)

- Notice for example:
  - R26 and XL are meant to be the same register
  - R31 and ZH are meant to be the same register
  - etc.
  - (Usually, though, we need to use assembler directives to get such multiple-naming of registers...)
- AVR convention:
  - When using one of the three named register pairs for 16-bit numbers...
  - ... high byte is always in the odd-numbered register ...
  - ... while the low byte is always the even-numbered register.
  - We can never set X directly but only separately set XH (R27) and XL (R26)

Storing 0x77 in SRAM at address 0x200

```
LDI R16, 0x77
STS 0x200, R16
```

STS must have a destination address that is known at compile time (**direct** addressing).

That is, during the execution phase, the CPU has everything it needs to know to immediately (directly) access the memory location referred to in the STS instruction.

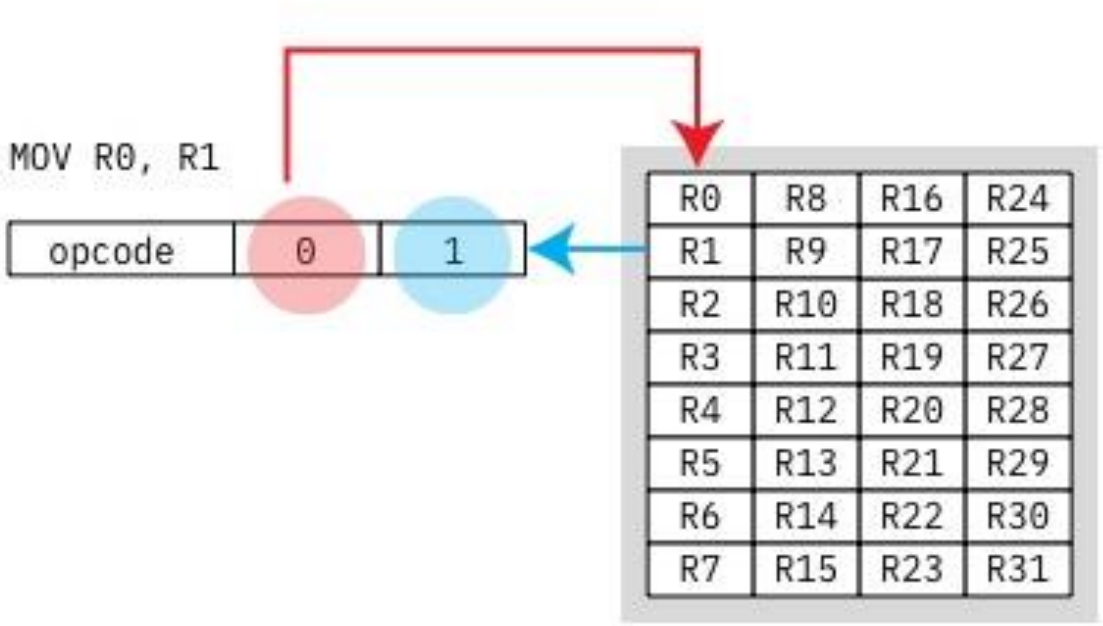
```
LDI R16, 0x77
LDI R27, HIGH(0x200)
LDI R26, LOW(0x200)
ST  X, R16
```

ST uses an address that may be constructed / computed at run time (**indirect** addressing).

That is, during the execution phase, the CPU must first read the contents of XH:XL, and only then access the memory location referred to in the STS instruction.

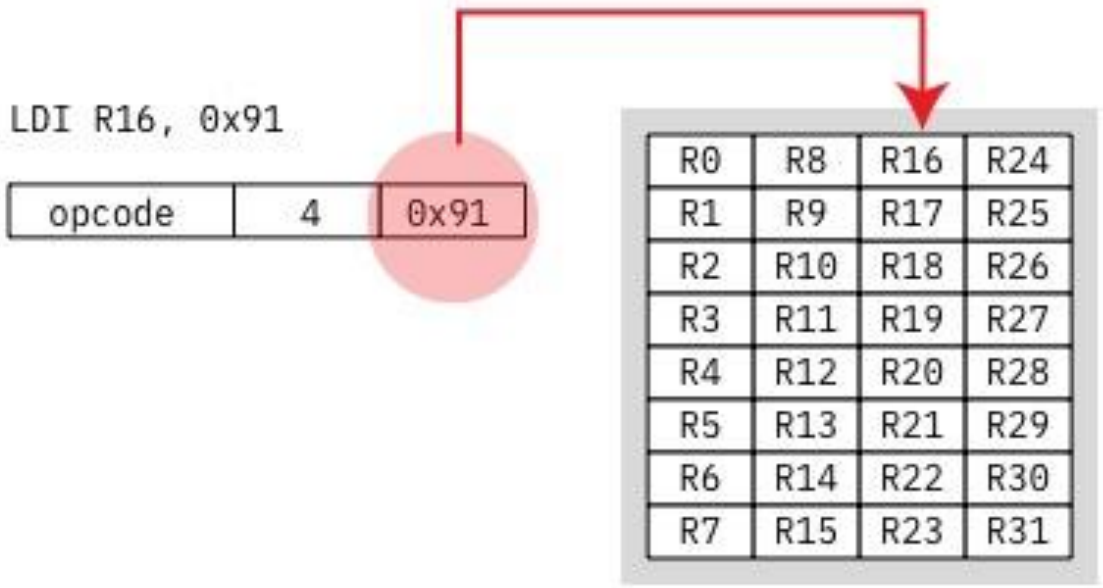
HIGH and LOW are assembler macros.

Direct addressing (reg to reg)



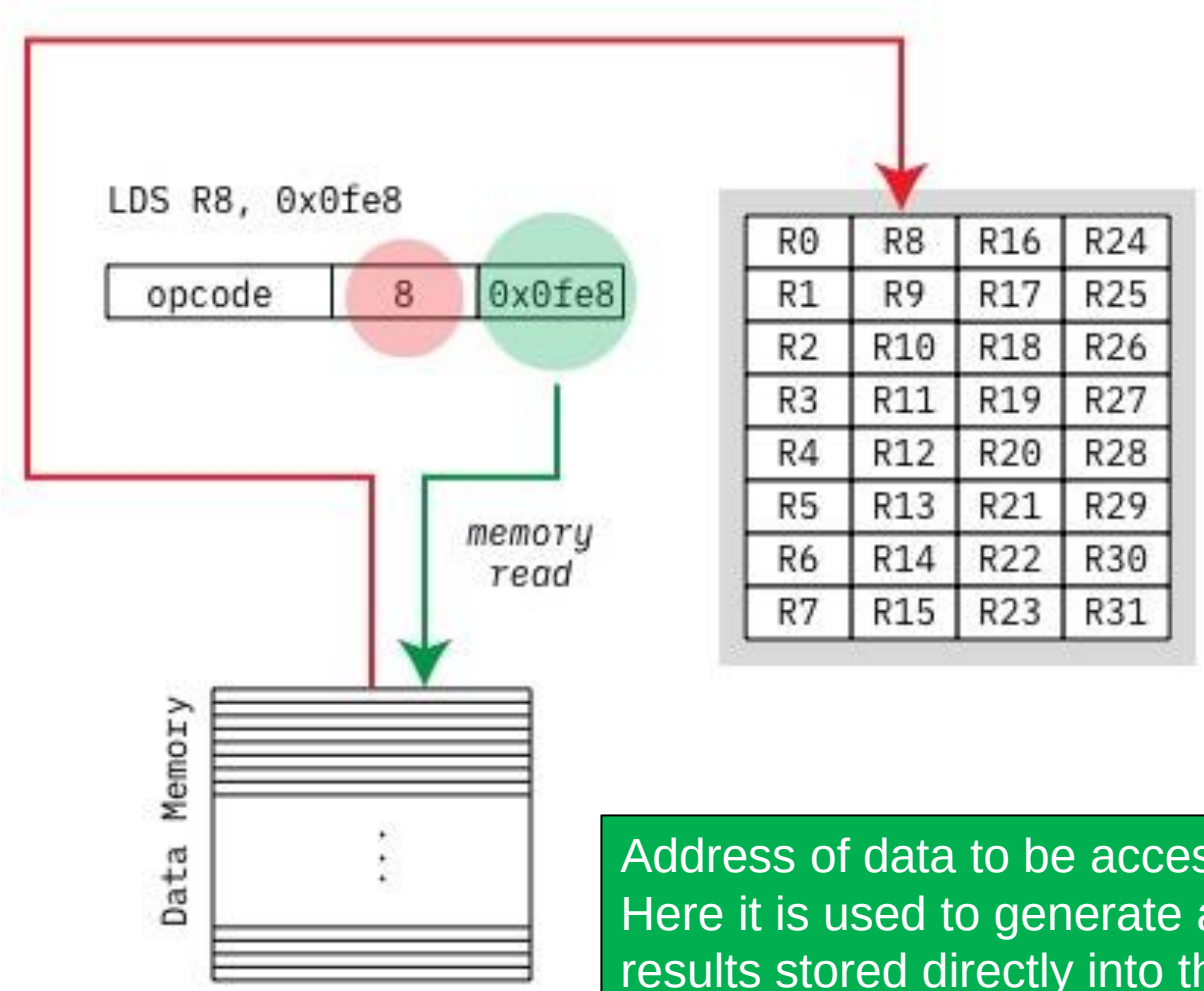
Value in source register is directly copied into destination register.

Direct addressing (literal to reg)



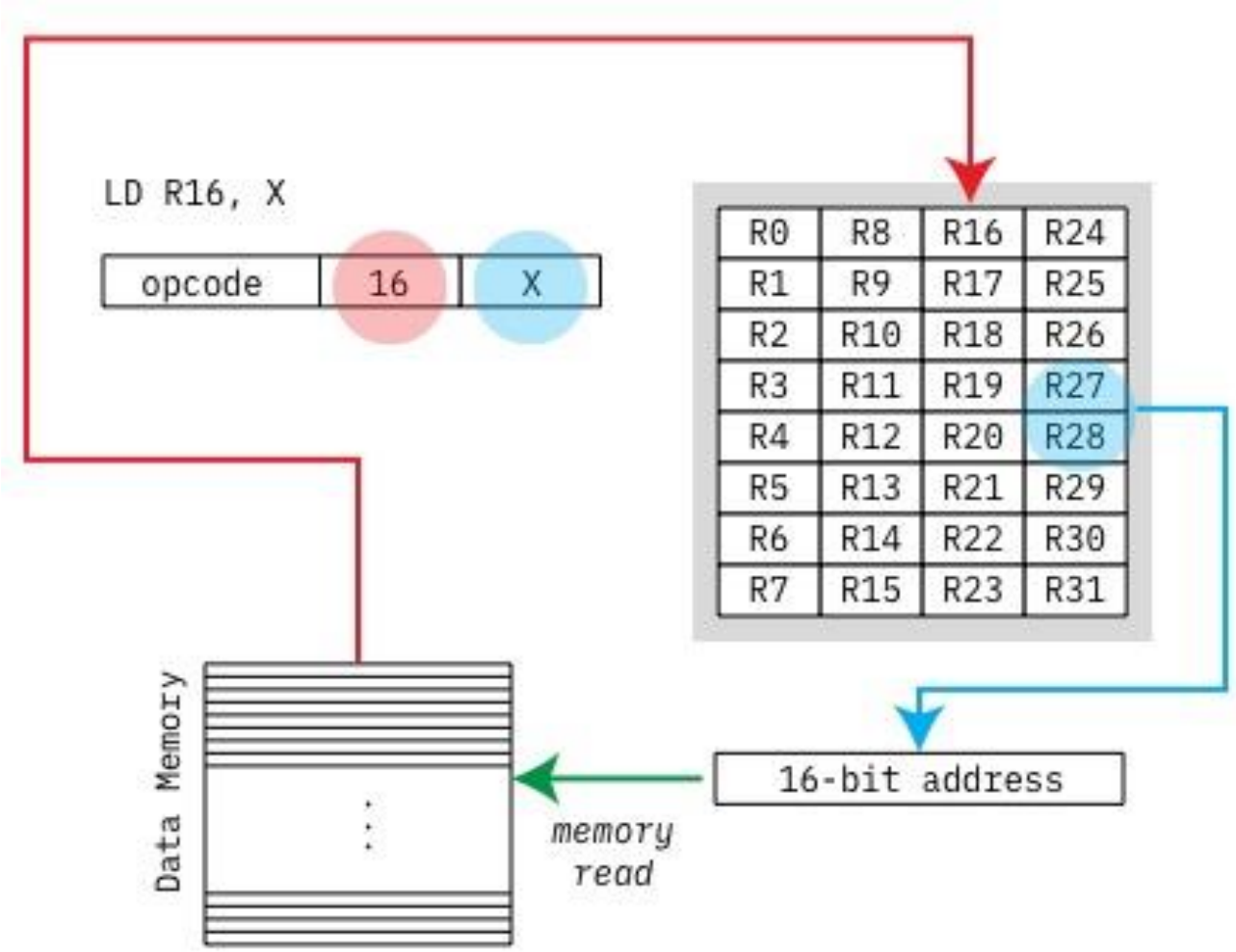
Value of constant in opcode is directly written to register in opcode.

Direct data addressing



Address of data to be accessed is within the opcode. Here it is used to generate a data memory read, with results stored directly into the destination register.

Indirect data addressing

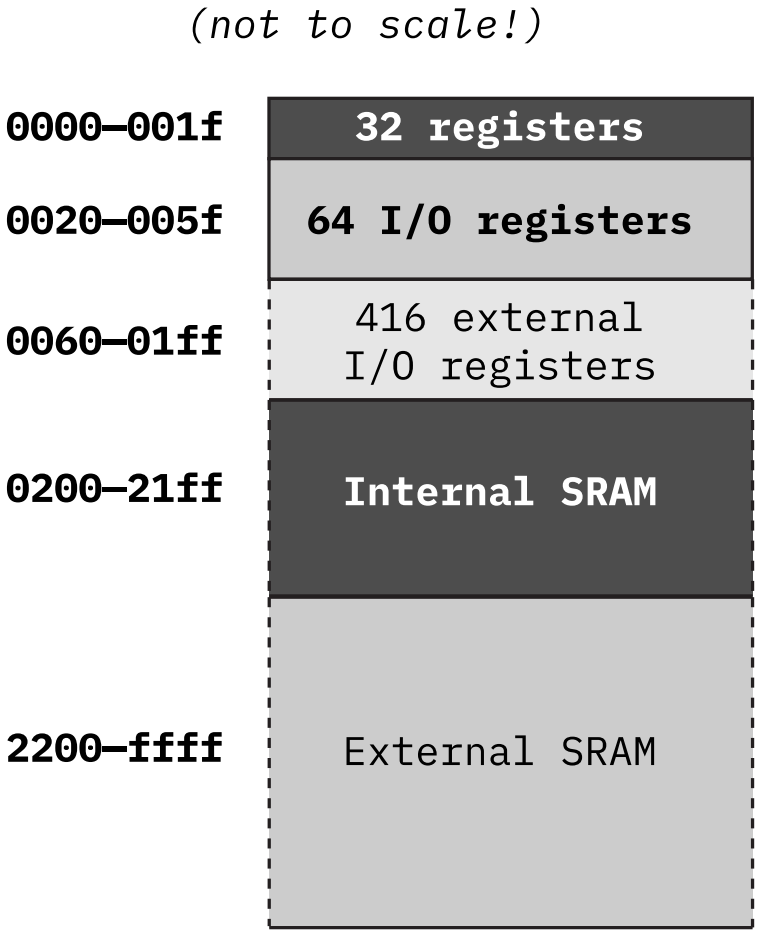


Pseudo register's content is read, and used as an address into data memory. Data memory is then read, and result stored into destination register. The blue indicates the indirection.

Addressing modes

- If we know the source and destination of data at compile time...
  - ... such as which register is source or destination ...
  - ... or the actual hardcoded address is given
  - ... then we can use direct addressing (e.g., LDI, MOV, LDS, STS)
- If the source or destination of a data address is only known at run time:
  - then we use indirect addressing (e.g., LD, ST)

Data Memory Map (mega2560)



Notice anything strange (i.e., more examples of some things having more than one name)?

SRAM 0x00 == R00 (!)

- Memory locations 0x00 to 0x1F exactly match registers R0 to R31
  - SRAM 0x01 is register R1...
  - SRAM 0x02 is register R2...
  - ... etc.
- The data-memory space serves many purposes
  - We must be crystal clear regarding what locations are available to us for our own data (i.e., internal SRAM)...
  - ... and which locations access other mega2560 resources

Any Questions?