CSC 230 - Summer 2018

Introduction to Assembly Language

Bill Bird

Department of Computer Science
University of Victoria

June 3, 2018

A Simple Machine Language (1)

Opcode	**Instruction**
000	Stop execution
001	Add memory value to R
010	Subtract memory value from R
011	Load from memory into R
100	Store R into memory
101	Set R to argument value

Consider a CPU with 1 register called R and a main memory with 32 locations, numbered 0 through 31. Each memory location, and the register R, can store 8 bits (or 1 **byte**). As a result, 8 bits is the 'natural' size of numbers for this processor. We say that the processor has an 8-bit **word size**.

A Simple Machine Language (2)

Opcode	**Instruction**
000	Stop execution
001	Add memory value to R
010	Subtract memory value from R
011	Load from memory into R
100	Store R into memory
101	Set R to argument value

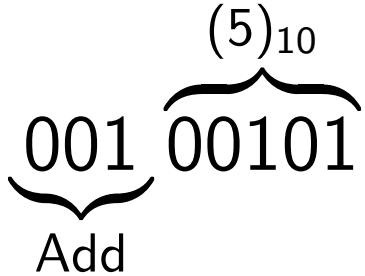
(Don't worry about knowing about this fictional architecture for exams; we will use AVR for everything after this example is over)

A Simple Machine Language (3)

Opcode	**Instruction**
000	Stop execution
001	Add memory value to R
010	Subtract memory value from R
011	Load from memory into R
100	Store R into memory
101	Set R to argument value

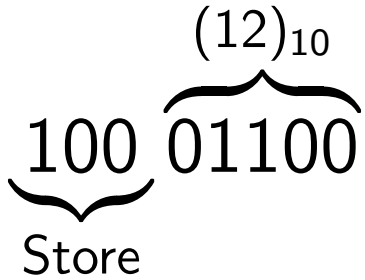
The machine code instructions are each 8 bits long, with the first three bits specifying an **opcode**, which corresponds to an operation to perform. The remaining five bits specify an argument in binary (either a memory address or a constant).

A Simple Machine Language (4)



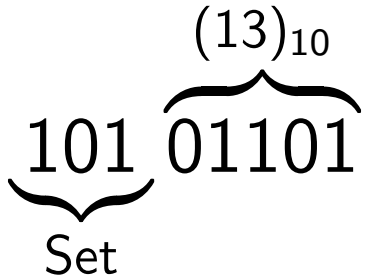
Opcode 001 is 'Add memory value to R'. The argument 00101 is the binary representation of 5. This instruction therefore translates to 'Add the contents of memory location 5 to R'

A Simple Machine Language (5)



Similarly, Opcode 100 is 'Store R into memory'. The argument 01100 is the binary representation of 12. This instruction therefore translates to 'Store the value in R into memory location 12'

A Simple Machine Language (6)



The 5-bit argument can also be used as a constant value. For example, the opcode 101 simply stores the number specified by the argument into R. The instruction above would set R to 13.

A Simple Machine Language (7)

Program

10100011
10000101
10111010
10000010
01100101
00100010
10000001
00000000

CPU

R

Memory

Address	Contents
0	0
1	0
2	0
3	0
4	0
5	0
6	0

A **program** is simply a sequence of machine instructions. Observe that the machine instructions (even in this simple architecture) are not very easy for humans to decipher.

A Simple Machine Language (8)

Program

10100011
10000101
10111010
10000010
01100101
00100010
10000001
00000000

CPU

R

Memory

Address	Contents
0	0
1	0
2	0
3	0
4	0
5	0
6	0

To execute the program, the CPU executes the sequence of instructions in order.

A Simple Machine Language (9)

Program

10100011
10000101
10111010
10000010
01100101
00100010
10000001
00000000

CPU

R

Memory

Address	Contents
0	0
1	0
2	0
3	0
4	0
5	0
6	0

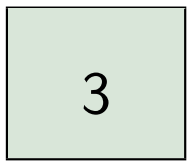
In the case where the instructions comprising the program are stored in memory, a special register called the **program counter** (PC) is used to track the address of the current instruction.

A Simple Machine Language (10)

Program

10100011
10000101
10111010
10000010
01100101
00100010
10000001
00000000

CPU



Memory

Address	Contents
0	0
1	0
2	0
3	0
4	0
5	0
6	0

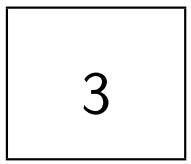
The instruction 10100011 has the opcode 101 (Set R) and the argument 00011 $= (3)_{10}$, so R is set to the value 3.

A Simple Machine Language (11)

Program

10100011
10000101
10111010
10000010
01100101
00100010
10000001
00000000

CPU



Memory

Address	Contents
0	0
1	0
2	0
3	0
4	0
5	3
6	0

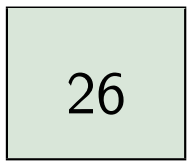
The instruction 10000101 has the opcode 100 (Store R into memory) and the argument $00101 = (5)_{10}$, so the value in R is stored to address 5.

A Simple Machine Language (12)

Program

10100011
10000101
10111010
10000010
01100101
00100010
10000001
00000000

CPU



Memory

Address	Contents
0	0
1	0
2	0
3	0
4	0
5	3
6	0

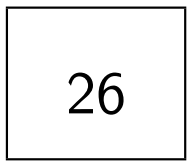
The instruction 10111010 has the opcode 101 (Set R) and the argument $11010 = (26)_{10}$, so R is set to 26.

A Simple Machine Language (13)

Program

10100011
10000101
10111010
10000010
01100101
00100010
10000001
00000000

CPU



Memory

Address	Contents
0	0
1	0
2	26
3	0
4	0
5	3
6	0

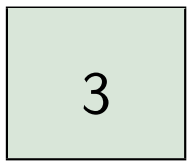
The instruction 10000010 has the opcode 100 (Store R into memory) and the argument $00010 = (2)_{10}$, so the value in R is stored to address 2.

A Simple Machine Language (14)

Program

```
10100011
10000101
10111010
10000010
01100101
00100010
10000001
00000000
```

CPU



Memory

Address	Contents
0	0
1	0
2	26
3	0
4	0
5	3
6	0

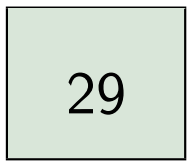
The instruction 01100101 has the opcode 011 (Load from memory into R) and the argument $00101 = (5)_{10}$, so the value in memory address 5 is loaded into R.

A Simple Machine Language (15)

Program

10100011
10000101
10111010
10000010
01100101
00100010
10000001
00000000

CPU



Memory

Address	Contents
0	0
1	0
2	26
3	0
4	0
5	3
6	0

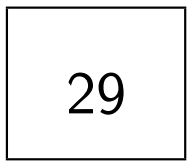
The instruction 00100010 has the opcode 001 (Add memory value to R) and the argument $00010 = (2)_{10}$, so the value in memory address 2 is added to R.

A Simple Machine Language (16)

Program

```
10100011
10000101
10111010
10000010
01100101
00100010
10000001
00000000
```

CPU



Memory

Address	Contents
0	0
1	29
2	26
3	0
4	0
5	3
6	0

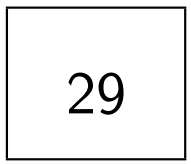
The instruction 10000001 has the opcode 100 (Store R into memory) and the argument $00001 = (1)_{10}$, so the value in R is stored to address 1.

A Simple Machine Language (17)

Program

```
10100011
10000101
10111010
10000010
01100101
00100010
10000001
00000000
```

CPU



Memory

Address	Contents
0	0
1	29
2	26
3	0
4	0
5	3
6	0

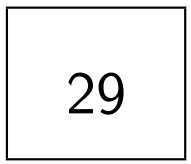
The instruction 00000000 has the opcode 000 (Stop execution), so the program terminates.

A Simple Machine Language (18)

Program

```
10100011
10000101
10111010
10000010
01100101
00100010
10000001
00000000
```

CPU



Memory

Address	Contents
0	0
1	29
2	26
3	0
4	0
5	3
6	0

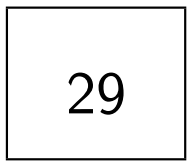
The programming process for early computers was painstaking, since programmers had to work with cryptic machine code and keep track of the usage and purpose of each memory location.

A Simple Machine Language (19)

Program

```
SET 00011
STORE 00101
SET 11010
STORE 00010
LOAD 00101
ADD 00010
STORE 00001
STOP 00000
```

CPU



Memory

Address	Contents
0	0
1	29
2	26
3	0
4	0
5	3
6	0

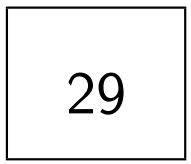
Idea: Instead of writing instructions as binary machine code, why not use a set of **mnemonics** to represent each opcode in human-readable terms?

A Simple Machine Language (20)

Program

```
SET 00011
STORE 00101
SET 11010
STORE 00010
LOAD 00101
ADD 00010
STORE 00001
STOP 00000
```

CPU



Memory

Address	Contents
0	0
1	29
2	26
3	0
4	0
5	3
6	0

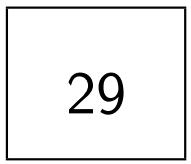
It is trivial to replace each mnemonic with its binary value, so writing instructions this way does not add any real complexity to the programming process.

A Simple Machine Language (21)

Program

```
SET 3
STORE 5
SET 26
STORE 2
LOAD 5
ADD 2
STORE 1
STOP 0
```

CPU



Memory

Address	Contents
0	0
1	29
2	26
3	0
4	0
5	3
6	0

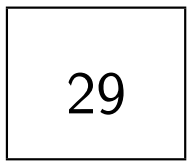
Idea: Instead of representing the argument values in binary, why not use decimal (or hexadecimal)?

A Simple Machine Language (22)

Program

```
SET 3
STORE 5
SET 26
STORE 2
LOAD 5
ADD 2
STORE 1
STOP 0
```

CPU



Memory

Address	Contents
0	0
1	29
2	26
3	0
4	0
5	3
6	0

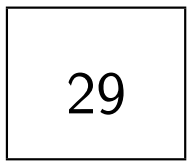
Again, since it is easy to convert from decimal to binary, it is easy to convert this representation back to binary machine code if necessary.

A Simple Machine Language (23)

Program

```
SET 3
STORE 5
SET 26
STORE 2
LOAD 5
ADD 2
STORE 1
STOP 0
```

CPU



Memory

Address	Contents
0	0
1	29
2	26
3	0
4	0
5	3
6	0

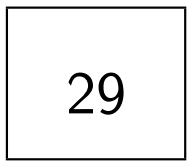
One issue which may not be obvious with small programs like the one above relates to **memory management**. The programmer must keep track of each memory location that they use, and make sure they don't use a memory location for two conflicting purposes.

A Simple Machine Language (24)

Program

```
SET 3
STORE Y
SET 26
STORE X
LOAD Y
ADD X
STORE Z
STOP 0
```

CPU



Memory

Address	Contents
0	0
1 (Z)	29
2 (X)	26
3	0
4	0
5 (Y)	3
6	0

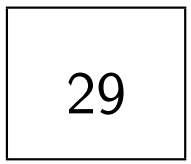
Idea: Instead of using numbers to refer to memory locations, why not give each location a name?

A Simple Machine Language (25)

Program

```
SET 3
STORE Y
SET 26
STORE X
LOAD Y
ADD X
STORE Z
STOP 0
```

CPU



Memory

Address	Contents
0	0
1 (Z)	29
2 (X)	26
3	0
4	0
5 (Y)	3
6	0

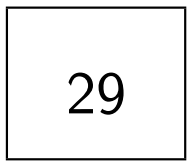
As long as we keep track of which locations have which names, it is still easy to convert this representation back to binary.

A Simple Machine Language (26)

Program

```
SET 3
STORE Y
SET 26
STORE X
LOAD Y
ADD X
STORE Z
STOP 0
```

CPU



Memory

Address	Contents
0	0
1 (Z)	29
2 (X)	26
3	0
4	0
5 (Y)	3
6	0

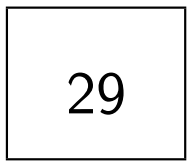
Using mnemonics for opcodes, allowing decimal or hex to be used instead of binary, and allowing memory locations to be named makes it much easier to assemble computer programs from machine instructions.

A Simple Machine Language (27)

Program

```
SET 3
STORE Y
SET 26
STORE X
LOAD Y
ADD X
STORE Z
STOP 0
```

CPU



Memory

Address	Contents
0	0
1 (Z)	29
2 (X)	26
3	0
4	0
5 (Y)	3
6	0

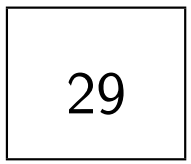
As a result, this representation is called **assembly language**. Since each processor might have a different set of instructions, the assembly language for each processor may be different.

A Simple Machine Language (28)

Program

```
SET 3
STORE Y
SET 26
STORE X
LOAD Y
ADD X
STORE Z
STOP 0
```

CPU



Memory

Address	Contents
0	0
1 (Z)	29
2 (X)	26
3	0
4	0
5 (Y)	3
6	0

Assembly languages are usually not considered to be programming languages, since they are only a thin abstraction of the machine language. Specifically, it is possible to convert from assembly to binary machine code and vice versa.

AVR Assembly Language (1)

Program Memory

```
(0x0000)    inc   r0
(0x0001)    dec   r1
(0x0002)    dec   r1
(0x0003)    mov   r2, r0
(0x0004)    sub   r2, r0
(0x0005)    sts   0x200, r1
(0x0007)    neg   r1
(0x0008)    sts   0x202, r1
(0x000a)    nop
```

Registers

r00	0x00
r01	0x00
r02	0x00
r03	0x00
r04	0x00

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

Data Memory

Address	Contents
0x0200	0x00
0x0201	0x00
0x0202	0x00
0x0203	0x00
0x0204	0x00
0x0205	0x00

The AVR architecture supports an instruction set of about 130 instructions. Instructions take between zero and two operands, which are usually registers. Some instructions (ending with 'I') work with 'immediate' numerical operands.

AVR Assembly Language (2)

Program Memory

```
(0x0000)    inc   r0
(0x0001)    dec   r1
(0x0002)    dec   r1
(0x0003)    mov   r2, r0
(0x0004)    sub   r2, r0
(0x0005)    sts   0x200, r1
(0x0007)    neg   r1
(0x0008)    sts   0x202, r1
(0x000a)    nop
```

Registers

r00	0x00
r01	0x00
r02	0x00
r03	0x00
r04	0x00

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

Data Memory

Address	Contents
0x0200	0x00
0x0201	0x00
0x0202	0x00
0x0203	0x00
0x0204	0x00
0x0205	0x00

The ATmega2560 has 32 registers, labelled `r0` through `r31`. Some instructions only work with a subset of registers. In particular, instructions taking immediate operands can only work with registers `r16` through `r31`.

AVR Assembly Language (3)

Program Memory

```
(0x0000)    inc   r0
(0x0001)    dec   r1
(0x0002)    dec   r1
(0x0003)    mov   r2, r0
(0x0004)    sub   r2, r0
(0x0005)    sts   0x200, r1
(0x0007)    neg   r1
(0x0008)    sts   0x202, r1
(0x000a)    nop
```

Registers

r00	0x00
r01	0x00
r02	0x00
r03	0x00
r04	0x00

C	N	Z	V

Data Memory

Address	Contents
0x0200	0x00
0x0201	0x00
0x0202	0x00
0x0203	0x00
0x0204	0x00
0x0205	0x00

Each instruction is encoded into a 16 or 32 bit binary representation and stored in **program memory**. Normally, the first instruction to execute is stored at address 0x0000 in program memory.

AVR Assembly Language (4)

Program Memory

(0x0000)	`inc`	`r0`
(0x0001)	`dec`	`r1`
(0x0002)	`dec`	`r1`
(0x0003)	`mov`	`r2, r0`
(0x0004)	`sub`	`r2, r0`
(0x0005)	`sts`	`0x200, r1`
(0x0007)	`neg`	`r1`
(0x0008)	`sts`	`0x202, r1`
(0x000a)	`nop`	

Registers

r00	0x00
r01	0x00
r02	0x00
r03	0x00
r04	0x00

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

Data Memory

Address	Contents
0x0200	0x00
0x0201	0x00
0x0202	0x00
0x0203	0x00
0x0204	0x00
0x0205	0x00

Each location in program memory stores a 16-bit value, so the entire `inc r0` instruction (which is 16 bits in size) can fit into address 0x0000. Some instructions (like the `sts` instructions) require two consecutive slots in program memory.

AVR Assembly Language (5)

Program Memory

```
(0x0000)    inc   r0
(0x0001)    dec   r1
(0x0002)    dec   r1
(0x0003)    mov   r2, r0
(0x0004)    sub   r2, r0
(0x0005)    sts   0x200, r1
(0x0007)    neg   r1
(0x0008)    sts   0x202, r1
(0x000a)    nop
```

Registers

r00	0x00
r01	0x00
r02	0x00
r03	0x00
r04	0x00

C	N	Z	V

Data Memory

Address	Contents
0x0200	0x00
0x0201	0x00
0x0202	0x00
0x0203	0x00
0x0204	0x00
0x0205	0x00

A special register called the **program counter** (PC) is used to track the address of the next instruction to execute in program memory. In these slides, the next instruction will be indicated by green highlighting.

AVR Assembly Language (6)

Program Memory

```
(0x0000)    inc   r0
(0x0001)    dec   r1
(0x0002)    dec   r1
(0x0003)    mov   r2, r0
(0x0004)    sub   r2, r0
(0x0005)    sts   0x200, r1
(0x0007)    neg   r1
(0x0008)    sts   0x202, r1
(0x000a)    nop
```

Registers

r00	0x00
r01	0x00
r02	0x00
r03	0x00
r04	0x00

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

Data Memory

Address	Contents
0x0200	0x00
0x0201	0x00
0x0202	0x00
0x0203	0x00
0x0204	0x00
0x0205	0x00

(To be clear: the highlighted line in the examples on these slides will always be the **next** instruction to run, not the instruction that was just run)

AVR Assembly Language (7)

Program Memory

```
(0x0000)    inc   r0
(0x0001)    dec   r1
(0x0002)    dec   r1
(0x0003)    mov   r2, r0
(0x0004)    sub   r2, r0
(0x0005)    sts   0x200, r1
(0x0007)    neg   r1
(0x0008)    sts   0x202, r1
(0x000a)    nop
```

Registers

r00	0x00
r01	0x00
r02	0x00
r03	0x00
r04	0x00

C	N	Z	V

Data Memory

Address	Contents
0x0200	0x00
0x0201	0x00
0x0202	0x00
0x0203	0x00
0x0204	0x00
0x0205	0x00

General purpose storage is available in **data memory**, in which each address refers to an 8-bit value (unlike program memory which uses 16 bit per location). For reasons that will become clear later, the lowest general-purpose address in data memory is 0x200.

AVR Assembly Language (8)

Program Memory

```
(0x0000)    inc   r0
(0x0001)    dec   r1
(0x0002)    dec   r1
(0x0003)    mov   r2, r0
(0x0004)    sub   r2, r0
(0x0005)    sts   0x200, r1
(0x0007)    neg   r1
(0x0008)    sts   0x202, r1
(0x000a)    nop
```

Registers

r00	0x01
r01	0x00
r02	0x00
r03	0x00
r04	0x00

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

Data Memory

Address	Contents
0x0200	0x00
0x0201	0x00
0x0202	0x00
0x0203	0x00
0x0204	0x00
0x0205	0x00

The `inc` instruction increments the specified register.

AVR Assembly Language (9)

Program Memory

```
(0x0000)    inc   r0
(0x0001)    dec   r1
(0x0002)    dec   r1
(0x0003)    mov   r2, r0
(0x0004)    sub   r2, r0
(0x0005)    sts   0x200, r1
(0x0007)    neg   r1
(0x0008)    sts   0x202, r1
(0x000a)    nop
```

Registers

r00	0x01
r01	0xff
r02	0x00
r03	0x00
r04	0x00

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

Data Memory

Address	Contents
0x0200	0x00
0x0201	0x00
0x0202	0x00
0x0203	0x00
0x0204	0x00
0x0205	0x00

The `dec` instruction decrements a register. Notice that the resulting value `0xff` can be interpreted either as 255 or -1 (and the choice of interpretation is up to the programmer).

AVR Assembly Language (10)

Program Memory

```
(0x0000)    inc   r0
(0x0001)    dec   r1
(0x0002)    dec   r1
(0x0003)    mov   r2, r0
(0x0004)    sub   r2, r0
(0x0005)    sts   0x200, r1
(0x0007)    neg   r1
(0x0008)    sts   0x202, r1
(0x000a)    nop
```

Registers

r00	0x01
r01	0xff
r02	0x00
r03	0x00
r04	0x00

C N Z V

Data Memory

Address	Contents
0x0200	0x00
0x0201	0x00
0x0202	0x00
0x0203	0x00
0x0204	0x00
0x0205	0x00

The processor uses a special **status register** (or SREG) to keep track of certain conditions resulting from arithmetic operations. When arithmetic instructions like `inc`, `dec`, `add` or `sub` are performed, one or more **flags** will be set in SREG.

AVR Assembly Language (11)

Program Memory

```
(0x0000)    inc   r0
(0x0001)    dec   r1
(0x0002)    dec   r1
(0x0003)    mov   r2, r0
(0x0004)    sub   r2, r0
(0x0005)    sts   0x200, r1
(0x0007)    neg   r1
(0x0008)    sts   0x202, r1
(0x000a)    nop
```

Registers

r00	0x01
r01	0xff
r02	0x00
r03	0x00
r04	0x00

C **N** Z V

Data Memory

Address	Contents
0x0200	0x00
0x0201	0x00
0x0202	0x00
0x0203	0x00
0x0204	0x00
0x0205	0x00

The `dec` instruction above produced the result `0xff`, which is negative in two's complement. Therefore, the N (Negative) flag was set in SREG. Flag values are used by control flow logic (among other instructions).

AVR Assembly Language (12)

Program Memory

```
(0x0000)    inc   r0
(0x0001)    dec   r1
(0x0002)    dec   r1
(0x0003)    mov   r2, r0
(0x0004)    sub   r2, r0
(0x0005)    sts   0x200, r1
(0x0007)    neg   r1
(0x0008)    sts   0x202, r1
(0x000a)    nop
```

Registers

r00	0x01
r01	0xff
r02	0x00
r03	0x00
r04	0x00

C N Z V

Data Memory

Address	Contents
0x0200	0x00
0x0201	0x00
0x0202	0x00
0x0203	0x00
0x0204	0x00
0x0205	0x00

The flags shown above are Carry (C), Negative (N), Zero (Z) and Two's Complement Overflow (V). The SREG register also contains four other flags which are less useful for the moment. We will cover SREG in extreme detail in a future lecture.

AVR Assembly Language (13)

Program Memory

```
(0x0000)    inc   r0
(0x0001)    dec   r1
(0x0002)    dec   r1
(0x0003)    mov   r2, r0
(0x0004)    sub   r2, r0
(0x0005)    sts   0x200, r1
(0x0007)    neg   r1
(0x0008)    sts   0x202, r1
(0x000a)    nop
```

Registers

r00	0x01
r01	0xff
r02	0x00
r03	0x00
r04	0x00

C	N	Z	V

Data Memory

Address	Contents
0x0200	0x00
0x0201	0x00
0x0202	0x00
0x0203	0x00
0x0204	0x00
0x0205	0x00

Each instruction will set zero or more of the flags in SREG. The various instruction references list the flags set by each instruction.

AVR Assembly Language (14)

Program Memory

```
(0x0000)    inc   r0
(0x0001)    dec   r1
(0x0002)    dec   r1
(0x0003)    mov   r2, r0
(0x0004)    sub   r2, r0
(0x0005)    sts   0x200, r1
(0x0007)    neg   r1
(0x0008)    sts   0x202, r1
(0x000a)    nop
```

Registers

r00	0x01
r01	0xfe
r02	0x00
r03	0x00
r04	0x00

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

Data Memory

Address	Contents
0x0200	0x00
0x0201	0x00
0x0202	0x00
0x0203	0x00
0x0204	0x00
0x0205	0x00

The second **dec** instruction also has a negative result (which sets the N flag again).

AVR Assembly Language (15)

Program Memory

```
(0x0000)    inc   r0
(0x0001)    dec   r1
(0x0002)    dec   r1
(0x0003)    mov   r2, r0
(0x0004)    sub   r2, r0
(0x0005)    sts   0x200, r1
(0x0007)    neg   r1
(0x0008)    sts   0x202, r1
(0x000a)    nop
```

Registers

r00	0x01
r01	0xfe
r02	0x01
r03	0x00
r04	0x00

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

Data Memory

Address	Contents
0x0200	0x00
0x0201	0x00
0x0202	0x00
0x0203	0x00
0x0204	0x00
0x0205	0x00

Since the `mov` instruction (which copies a value from one register to another) is not listed as setting the N flag, the flag remains set after the `mov` instruction runs.

AVR Assembly Language (16)

Program Memory

```
(0x0000)    inc   r0
(0x0001)    dec   r1
(0x0002)    dec   r1
(0x0003)    mov   r2, r0
(0x0004)    sub   r2, r0
(0x0005)    sts   0x200, r1
(0x0007)    neg   r1
(0x0008)    sts   0x202, r1
(0x000a)    nop
```

Registers

r00	0x01
r01	0xfe
r02	0x00
r03	0x00
r04	0x00

C N **Z** V

Data Memory

Address	Contents
0x0200	0x00
0x0201	0x00
0x0202	0x00
0x0203	0x00
0x0204	0x00
0x0205	0x00

The `sub` instruction above subtracts `r0` from `r2`, then stores the result back in `r2`. The `sub` instruction sets all four flags above. In this case, the result is zero, so the Z flag is set and the remaining flags are cleared.

AVR Assembly Language (17)

Program Memory

```
(0x0000)    inc   r0
(0x0001)    dec   r1
(0x0002)    dec   r1
(0x0003)    mov   r2, r0
(0x0004)    sub   r2, r0
(0x0005)    sts   0x200, r1
(0x0007)    neg   r1
(0x0008)    sts   0x202, r1
(0x000a)    nop
```

Registers

r00	0x01
r01	0xfe
r02	0x00
r03	0x00
r04	0x00

C	N	Z	V

Data Memory

Address	Contents
0x0200	0xfe
0x0201	0x00
0x0202	0x00
0x0203	0x00
0x0204	0x00
0x0205	0x00

The `sts` instruction stores the value in the specified register into data memory at the specified location. Note that `sts` does not modify any of the active flags (regardless of the value of the specified register).

AVR Assembly Language (18)

Program Memory

```
(0x0000)    inc   r0
(0x0001)    dec   r1
(0x0002)    dec   r1
(0x0003)    mov   r2, r0
(0x0004)    sub   r2, r0
(0x0005)    sts   0x200, r1
(0x0007)    neg   r1
(0x0008)    sts   0x202, r1
(0x000a)    nop
```

Registers

r00	0x01
r01	0x02
r02	0x00
r03	0x00
r04	0x00

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

Data Memory

Address	Contents
0x0200	0xfe
0x0201	0x00
0x0202	0x00
0x0203	0x00
0x0204	0x00
0x0205	0x00

The `neg` instruction above negates `r1` (by computing the expression $0 - \texttt{r1}$). The result is positive and nonzero, so the Z flag is cleared. The carry flag is set, however, since the subtraction $0 - \texttt{r1}$ results in a carry.

AVR Assembly Language (19)

Program Memory

```
(0x0000)    inc   r0
(0x0001)    dec   r1
(0x0002)    dec   r1
(0x0003)    mov   r2, r0
(0x0004)    sub   r2, r0
(0x0005)    sts   0x200, r1
(0x0007)    neg   r1
(0x0008)    sts   0x202, r1
(0x000a)    nop
```

Registers

r00	0x01
r01	0x02
r02	0x00
r03	0x00
r04	0x00

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

Data Memory

Address	Contents
0x0200	0xfe
0x0201	0x00
0x0202	0x02
0x0203	0x00
0x0204	0x00
0x0205	0x00

Finally, the last `sts` instruction stores another value into memory.

AVR Assembly Language (20)

Program Memory

```
(0x0000)    inc   r0
(0x0001)    dec   r1
(0x0002)    dec   r1
(0x0003)    mov   r2, r0
(0x0004)    sub   r2, r0
(0x0005)    sts   0x200, r1
(0x0007)    neg   r1
(0x0008)    sts   0x202, r1
(0x000a)    nop
```

Registers

r00	0x01
r01	0x02
r02	0x00
r03	0x00
r04	0x00

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

Data Memory

Address	Contents
0x0200	0xfe
0x0201	0x00
0x0202	0x02
0x0203	0x00
0x0204	0x00
0x0205	0x00

The `nop` instruction has no effect (the instruction results in the processor doing nothing for one clock cycle, then continuing to the next instruction).

AVR Assembly Language (21)

Program Memory

```
(0x0000)    inc   r0
(0x0001)    dec   r1
(0x0002)    dec   r1
(0x0003)    mov   r2, r0
(0x0004)    sub   r2, r0
(0x0005)    sts   0x200, r1
(0x0007)    neg   r1
(0x0008)    sts   0x202, r1
(0x000a)    nop
```

Registers

r00	0x01
r01	0x02
r02	0x00
r03	0x00
r04	0x00

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

Data Memory

Address	Contents
0x0200	0xfe
0x0201	0x00
0x0202	0x02
0x0203	0x00
0x0204	0x00
0x0205	0x00

Question: What will happen if the program counter reaches a position past the last instruction in the listing above?

Infinite Loops (1)

Program Memory

```
(0x0000)    ldi   r16, 10
(0x0001)    ldi   r17, 20
(0x0002)    add   r16, r17
          done:
(0x0003)    jmp   done
```

Registers

r16	0x00
r17	0x00
r18	0x00
r19	0x00
r20	0x00

C	N	Z	V

Data Memory

Address	Contents
0x0200	0x00
0x0201	0x00
0x0202	0x00
0x0203	0x00
0x0204	0x00
0x0205	0x00

The program counter will continue to increment and run instructions indefinitely, since the contents of program memory is just binary data (and the processor has no way of knowing when our program ends).

Infinite Loops (2)

Program Memory

```
(0x0000)    ldi   r16, 10
(0x0001)    ldi   r17, 20
(0x0002)    add   r16, r17
          done:
(0x0003)    jmp   done
```

Registers

r16	0x00
r17	0x00
r18	0x00
r19	0x00
r20	0x00

C	N	Z	V

Data Memory

Address	Contents
0x0200	0x00
0x0201	0x00
0x0202	0x00
0x0203	0x00
0x0204	0x00
0x0205	0x00

If the program counter continues past the end of the program, whatever happens to be in the following positions of memory will be executed.

Infinite Loops (3)

Program Memory

```
(0x0000)    ldi   r16, 10
(0x0001)    ldi   r17, 20
(0x0002)    add   r16, r17
          done:
(0x0003)    jmp   done
```

Registers

r16	0x00
r17	0x00
r18	0x00
r19	0x00
r20	0x00

C	N	Z	V

Data Memory

Address	Contents
0x0200	0x00
0x0201	0x00
0x0202	0x00
0x0203	0x00
0x0204	0x00
0x0205	0x00

There is no 'stop' or 'end' instruction in the AVR instruction set. As long as the CPU has power, it will continue executing instructions.

Infinite Loops (4)

Program Memory

```
(0x0000)    ldi   r16, 10
(0x0001)    ldi   r17, 20
(0x0002)    add   r16, r17
          done:
(0x0003)    jmp   done
```

Registers

r16	0x00
r17	0x00
r18	0x00
r19	0x00
r20	0x00

C	N	Z	V

Data Memory

Address	Contents
0x0200	0x00
0x0201	0x00
0x0202	0x00
0x0203	0x00
0x0204	0x00
0x0205	0x00

There are special instructions to pause execution by changing the processor's state, but nothing to end execution altogether. For example, the `sleep` instruction will put the processor into sleep mode, but sleep mode is not permanent.

Infinite Loops (5)

Program Memory

```
(0x0000)    ldi   r16, 10
(0x0001)    ldi   r17, 20
(0x0002)    add   r16, r17
          done:
(0x0003)    jmp   done
```

Registers

r16	0x00
r17	0x00
r18	0x00
r19	0x00
r20	0x00

C	N	Z	V

Data Memory

Address	Contents
0x0200	0x00
0x0201	0x00
0x0202	0x00
0x0203	0x00
0x0204	0x00
0x0205	0x00

Instead of ending programs with a 'stop' instruction, we will create a deliberate infinite loop. The `jmp done` instruction above jumps (or **branches**) to the location labelled '`done:`', which effectively creates an infinite loop.

Infinite Loops (6)

Program Memory

```
(0x0000)    ldi   r16, 10
(0x0001)    ldi   r17, 20
(0x0002)    add   r16, r17
        done:
(0x0003)    jmp   done
```

Registers

r16	0x0a
r17	0x00
r18	0x00
r19	0x00
r20	0x00

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

Data Memory

Address	Contents
0x0200	0x00
0x0201	0x00
0x0202	0x00
0x0203	0x00
0x0204	0x00
0x0205	0x00

The `ldi` instruction loads an 8-bit constant into the specified register. Due to encoding constraints, only registers `r16` - `r31` can be used with `ldi`.

Infinite Loops (7)

Program Memory

```
(0x0000)    ldi   r16, 10
(0x0001)    ldi   r17, 20
(0x0002)    add   r16, r17
        done:
(0x0003)    jmp   done
```

Registers

r16	0x0a
r17	0x14
r18	0x00
r19	0x00
r20	0x00

C	N	Z	V

Data Memory

Address	Contents
0x0200	0x00
0x0201	0x00
0x0202	0x00
0x0203	0x00
0x0204	0x00
0x0205	0x00

The `ldi` instruction loads an 8-bit constant into the specified register. Due to encoding constraints, only registers `r16` - `r31` can be used with `ldi`.

Infinite Loops (8)

Program Memory

```
(0x0000)    ldi   r16, 10
(0x0001)    ldi   r17, 20
(0x0002)    add   r16, r17
        done:
(0x0003)    jmp   done
```

Registers

r16	0x1e
r17	0x14
r18	0x00
r19	0x00
r20	0x00

C	N	Z	V

Data Memory

Address	Contents
0x0200	0x00
0x0201	0x00
0x0202	0x00
0x0203	0x00
0x0204	0x00
0x0205	0x00

Infinite Loops (9)

Program Memory

```
(0x0000)    ldi   r16, 10
(0x0001)    ldi   r17, 20
(0x0002)    add   r16, r17
        done:
(0x0003)    jmp   done
```

Registers

r16	0x1e
r17	0x14
r18	0x00
r19	0x00
r20	0x00

C	N	Z	V

Data Memory

Address	Contents
0x0200	0x00
0x0201	0x00
0x0202	0x00
0x0203	0x00
0x0204	0x00
0x0205	0x00

Once the `jmp done` instruction is reached, the program counter will be fixed at `0x0003` permanently.

Infinite Loops (10)

Program Memory

```
(0x0000)    ldi   r16, 10
(0x0001)    ldi   r17, 20
(0x0002)    add   r16, r17
        done:
(0x0003)    jmp   done
```

Registers

r16	0x1e
r17	0x14
r18	0x00
r19	0x00
r20	0x00

C	N	Z	V

Data Memory

Address	Contents
0x0200	0x00
0x0201	0x00
0x0202	0x00
0x0203	0x00
0x0204	0x00
0x0205	0x00

(The `rjmp` instruction can be used instead of `jmp` in this case, and is used in many of the posted examples. In general, `rjmp` is useable in all cases where the jump destination is within 4096 addresses of the instruction)

Data Memory and Variables (1)

Program Memory

```
(0x0000)    ldi   r16, 230
(0x0001)    sts   MY_VAR, r16
(0x0003)    ldi   r16, 225
(0x0004)    sts   0x204, r16
          done:
(0x0006)    rjmp done
          .dseg
          .org 0x200
          MY_VAR: .byte 1
```

Registers

r16	0xe1
r17	0x00
r18	0x00
r19	0x00
r20	0x00

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

Data Memory

Address	Contents
0x0200	0xe6
0x0201	0x00
0x0202	0x00
0x0203	0x00
0x0204	0xe1
0x0205	0x00

Although memory locations can be manually encoded into instructions (as at address 0x0004 above), this can be error prone, and it is more practical to allocate named memory locations.

Data Memory and Variables (2)

Program Memory

```
(0x0000)    ldi   r16, 230
(0x0001)    sts   MY_VAR, r16
(0x0003)    ldi   r16, 225
(0x0004)    sts   0x204, r16
          done:
(0x0006)    rjmp done
          .dseg
          .org 0x200
          MY_VAR: .byte 1
```

Registers

r16	0xe1
r17	0x00
r18	0x00
r19	0x00
r20	0x00

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

Data Memory

Address	Contents
0x0200	0xe6
0x0201	0x00
0x0202	0x00
0x0203	0x00
0x0204	0xe1
0x0205	0x00

The `.dseg` directive above tells the assembler to work with data memory for the following lines (instead of program memory, which can be selected with a `.cseg` directive). The `.org 0x200` directive positions the assembler at address 0x200 in data memory.

Data Memory and Variables (3)

Program Memory

```
(0x0000)    ldi   r16, 230
(0x0001)    sts   MY_VAR, r16
(0x0003)    ldi   r16, 225
(0x0004)    sts   0x204, r16
          done:
(0x0006)    rjmp done
          .dseg
          .org 0x200
          MY_VAR: .byte 1
```

Registers

r16	0xe1
r17	0x00
r18	0x00
r19	0x00
r20	0x00

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

Data Memory

Address	Contents
0x0200	0xe6
0x0201	0x00
0x0202	0x00
0x0203	0x00
0x0204	0xe1
0x0205	0x00

The `.byte 1` directive instructs the assembler to set aside (i.e. skip) one byte of data memory. Since the assembler is positioned at address 0x200, the address of the allocated byte will be 0x200.

Data Memory and Variables (4)

Program Memory

```
(0x0000)    ldi   r16, 230
(0x0001)    sts   MY_VAR, r16
(0x0003)    ldi   r16, 225
(0x0004)    sts   0x204, r16
          done:
(0x0006)    rjmp done
          .dseg
          .org 0x200
          MY_VAR: .byte 1
```

Registers

r16	0xe1
r17	0x00
r18	0x00
r19	0x00
r20	0x00

C	N	Z	V

Data Memory

Address	Contents
0x0200	0xe6
0x0201	0x00
0x0202	0x00
0x0203	0x00
0x0204	0xe1
0x0205	0x00

Finally, attaching a label to any line of assembly (code or data) is equivalent to defining a numerical constant whose value is equal to the address of that line. In the code above, the label '`MY_VAR`' refers to address 0x200 in data memory, whereas the label `done` refers to address 0x0006 in program memory.

Data Memory and Variables (5)

Program Memory

```
(0x0000)    ldi  r16, 230
(0x0001)    sts  MY_VAR, r16
(0x0003)    ldi  r16, 225
(0x0004)    sts  0x204, r16
          done:
(0x0006)    rjmp done
          .dseg
          .org 0x200
          MY_VAR: .byte 1
```

Registers

r16	0xe1
r17	0x00
r18	0x00
r19	0x00
r20	0x00

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

Data Memory

Address	Contents
0x0200	0xe6
0x0201	0x00
0x0202	0x00
0x0203	0x00
0x0204	0xe1
0x0205	0x00

When a label is used in the assembly code, the assembler replaces its name with its numerical value. Therefore, the instruction '`sts MY_VAR, r16`' above is replaced by '`sts 0x200, r16`' when the code is assembled.

Data Memory and Variables (6)

Program Memory

```
(0x0000)    ldi   r16, 230
(0x0001)    sts   VAR1, r16
(0x0003)    ldi   r16, 225
(0x0004)    sts   VAR2, r16
          done:
(0x0006)    rjmp done
          .dseg
          .org 0x200
          VAR1:  .byte 1
          VAR2:  .byte 1
```

Registers

r16	0xe1
r17	0x00
r18	0x00
r19	0x00
r20	0x00

C	N	Z	V

Data Memory

Address	Contents
0x0200	0xe6
0x0201	0xe1
0x0202	0x00
0x0203	0x00
0x0204	0x00
0x0205	0x00

When multiple variables are allocated, they are positioned sequentially in memory (unless a `.org` directive is used to reposition the assembler between directives). In the example above, the variable `VAR2` is positioned at address `0x201`.

Data Memory and Variables (7)

Program Memory

```
(0x0000)    ldi   r16, 230
(0x0001)    sts   VAR1, r16
(0x0003)    ldi   r16, 225
(0x0004)    sts   VAR2, r16
          done:
(0x0006)    rjmp done
          .dseg
          .org 0x200
          VAR1:   .byte 3
          VAR2:   .byte 1
```

Registers

r16	0xe1
r17	0x00
r18	0x00
r19	0x00
r20	0x00

C	N	Z	V

Data Memory

Address	Contents
0x0200	0xe6
0x0201	0x00
0x0202	0x00
0x0203	0xe1
0x0204	0x00
0x0205	0x00

The `.byte` directive can be used to allocate more than one byte at a time (for example, to create arrays). In the case above, `VAR1` is allocated to be three bytes long, but the label `VAR1` still refers to the address `0x200` (the beginning of the three-byte block). The address of `VAR2` is shifted to `0x203`.

Multi-Byte Operations (1)

Program Memory

```
(0x0000)    ldi  r16, 0xea
(0x0001)    ldi  r17, 0x59
(0x0002)    ldi  r19, 0xe1
(0x0003)    ldi  r20, 0x00
(0x0004)    add  r16, r19
(0x0005)    add  r17, r20
        done:
(0x0006)    rjmp done
```

Registers

r16	0x00
r17	0x00
r18	0x00
r19	0x00
r20	0x00

C	N	Z	V

Data Memory

Address	Contents
0x0200	0x00
0x0201	0x00
0x0202	0x00
0x0203	0x00
0x0204	0x00
0x0205	0x00

Question: If all registers and memory locations are 8 bits wide, how can we perform 16- or 32-bit arithmetic?

Multi-Byte Operations (2)

Task: Compute the sum of 23018 (`0x59ea`) and 225 (`0x00e1`).
The result will be 23243 (`0x5acb`)

Multi-Byte Operations (3)

Idea: Treat each operand as a 16-bit number and store each operand in a pair of 8-bit registers.

Multi-Byte Operations (4)

Program Memory

```
(0x0000)    ldi   r16, 0xea
(0x0001)    ldi   r17, 0x59
(0x0002)    ldi   r19, 0xe1
(0x0003)    ldi   r20, 0x00
(0x0004)    add   r16, r19
(0x0005)    add   r17, r20
          done:
(0x0006)    rjmp done
```

Registers

r16	0x00
r17	0x00
r18	0x00
r19	0x00
r20	0x00

C	N	Z	V

Data Memory

Address	Contents
0x0200	0x00
0x0201	0x00
0x0202	0x00
0x0203	0x00
0x0204	0x00
0x0205	0x00

Idea: Treat each operand as a 16-bit number and store each operand in a pair of 8-bit registers.

Multi-Byte Operations (5)

Program Memory

```
(0x0000)    ldi   r16, 0xea
(0x0001)    ldi   r17, 0x59
(0x0002)    ldi   r19, 0xe1
(0x0003)    ldi   r20, 0x00
(0x0004)    add   r16, r19
(0x0005)    add   r17, r20
          done:
(0x0006)    rjmp done
```

Registers

r16	0x00
r17	0x00
r18	0x00
r19	0x00
r20	0x00

C	N	Z	V

Data Memory

Address	Contents
0x0200	0x00
0x0201	0x00
0x0202	0x00
0x0203	0x00
0x0204	0x00
0x0205	0x00

In the code above, the value `0x59ea` is stored in `r16` and `r17`, using a **little-endian** representation (in which the low-order byte is stored first). We will use the notation `r17:r16` to refer to 16-bit register pairs in little-endian order.

Multi-Byte Operations (6)

Program Memory

```
(0x0000)    ldi   r16, 0xea
(0x0001)    ldi   r17, 0x59
(0x0002)    ldi   r19, 0xe1
(0x0003)    ldi   r20, 0x00
(0x0004)    add   r16, r19
(0x0005)    add   r17, r20
          done:
(0x0006)    rjmp done
```

Registers

r16	0x00
r17	0x00
r18	0x00
r19	0x00
r20	0x00

C	N	Z	V

Data Memory

Address	Contents
0x0200	0x00
0x0201	0x00
0x0202	0x00
0x0203	0x00
0x0204	0x00
0x0205	0x00

Similarly, the value `0x00e1` is stored in `r20:r19`.

Multi-Byte Operations (7)

Program Memory

```
(0x0000)    ldi   r16, 0xea
(0x0001)    ldi   r17, 0x59
(0x0002)    ldi   r19, 0xe1
(0x0003)    ldi   r20, 0x00
(0x0004)    add   r16, r19
(0x0005)    add   r17, r20
        done:
(0x0006)    rjmp done
```

Registers

r16	0xcb
r17	0x59
r18	0x00
r19	0xe1
r20	0x00

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

Data Memory

Address	Contents
0x0200	0x00
0x0201	0x00
0x0202	0x00
0x0203	0x00
0x0204	0x00
0x0205	0x00

However, when the operation completes, the value of `r17:r16` is `0x59cb` instead of `0x5acb`.

Multi-Byte Operations (8)

Program Memory

```
(0x0000)    ldi   r16, 0xea
(0x0001)    ldi   r17, 0x59
(0x0002)    ldi   r19, 0xe1
(0x0003)    ldi   r20, 0x00
(0x0004)    add   r16, r19
(0x0005)    add   r17, r20
        done:
(0x0006)    rjmp done
```

Registers

r16	0xcb
r17	0x59
r18	0x00
r19	0xe1
r20	0x00

C	N	Z	V

Data Memory

Address	Contents
0x0200	0x00
0x0201	0x00
0x0202	0x00
0x0203	0x00
0x0204	0x00
0x0205	0x00

The low byte (0xcb) of the sum is correct, but the high byte is off by one (0x59 instead of 0x5a). The fundamental issue is that when the two low bytes (0xea and 0xe1) were added, the 8-bit addition resulted in an overflow (which should have been carried over into the high byte).

Multi-Byte Operations (9)

Program Memory

```
(0x0000)    ldi   r16, 0xea
(0x0001)    ldi   r17, 0x59
(0x0002)    ldi   r19, 0xe1
(0x0003)    ldi   r20, 0x00
(0x0004)    add   r16, r19
(0x0005)    add   r17, r20
        done:
(0x0006)    rjmp done
```

Registers

r16	0xcb
r17	0x59
r18	0x00
r19	0xe1
r20	0x00

C **N** Z V

Data Memory

Address	Contents
0x0200	0x00
0x0201	0x00
0x0202	0x00
0x0203	0x00
0x0204	0x00
0x0205	0x00

If we look at the state of the processor immediately after the `add r16, r19` instruction, we can see that the carry (C) flag was set by the instruction.

Multi-Byte Operations (10)

Program Memory

```
(0x0000)    ldi   r16, 0xea
(0x0001)    ldi   r17, 0x59
(0x0002)    ldi   r19, 0xe1
(0x0003)    ldi   r20, 0x00
(0x0004)    add   r16, r19
(0x0005)    add   r17, r20
        done:
(0x0006)    rjmp done
```

Registers

r16	0xcb
r17	0x59
r18	0x00
r19	0xe1
r20	0x00

C	N	Z	V

Data Memory

Address	Contents
0x0200	0x00
0x0201	0x00
0x0202	0x00
0x0203	0x00
0x0204	0x00
0x0205	0x00

For the high byte of the sum to be correct, the carry from the low byte must be incorporated into the second add instruction.

Multi-Byte Operations (11)

Program Memory

```
(0x0000)    ldi   r16, 0xea
(0x0001)    ldi   r17, 0x59
(0x0002)    ldi   r19, 0xe1
(0x0003)    ldi   r20, 0x00
(0x0004)    add   r16, r19
(0x0005)    adc   r17, r20
        done:
(0x0006)    rjmp done
```

Registers

r16	0xcb
r17	0x5a
r18	0x00
r19	0xe1
r20	0x00

C	N	Z	V

Data Memory

Address	Contents
0x0200	0x00
0x0201	0x00
0x0202	0x00
0x0203	0x00
0x0204	0x00
0x0205	0x00

The remedy is the adc instruction, which adds the values of two registers along with the current state of the carry bit. Notice that the resultin r17 is now correct.

Multi-Byte Operations (12)

Program Memory

```
(0x0000)    ldi   r16, 0xea
(0x0001)    ldi   r17, 0x59
(0x0002)    ldi   r19, 0xe1
(0x0003)    ldi   r20, 0x00
(0x0004)    add   r16, r19
(0x0005)    adc   r17, r20
        done:
(0x0006)    rjmp done
```

Registers

r16	0xcb
r17	0x5a
r18	0x00
r19	0xe1
r20	0x00

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

Data Memory

Address	Contents
0x0200	0x00
0x0201	0x00
0x0202	0x00
0x0203	0x00
0x0204	0x00
0x0205	0x00

In general, higher-width arithmetic (16, 32 or 64 bits) can be performed with a normal `add` instruction for the lowest byte, followed by a series of `adc` instructions for the higher bytes (which propagate any carried bits through the computation).