CSC 230: AVR arithmetic

AVR arithmetic

- Overriding themes of this section:
  - Relationship between number representations and arithmetic operations
  - Aspects of AVR design that permits us to use numbers as we intend
- Adding & subtracting one-byte numbers
- Arithmetic with pointers
- Adding & subtracting two-byte and four-byte numbers

Example

- The AVR processor is able to perform addition, subtraction on 8-bit data (i.e., bytes)
  - However, CPU is not able to tell if the data is unsigned integers or signed two's-complement numbers
  - We will soon see how the status register helps here!
- More arithmetic-instruction examples:

```
ADD Rd, Rr        ; same as Rd = Rd + Rr
SUB Rd, Rr        ; same as Rd = Rd - Rr
SUBI Rd, n        ; same as Rd = Rd - n
; note that there is no ADDI operation!
```

Recall binary addition example

	carry 1	carry 1	carry 1			
11	0	1	0	1	1	
6	0	0	1	1	0	+
17	1	0	0	0	1	

Binary addition

- One perspective: addition is performed on pairs of individual bits
  - The sum is the result of adding the bits
  - The carry is necessary if both bits are true (i.e., 1)
  - And do we consider an incoming carry from another pair of bits?
- Half-adder: takes two incoming bits A & B
- Full-adder: takes three incoming bits A, B and carry bit C
- Constructing hardware for an ADD instruction requires at least one half-adder and seven full adders.

Boolean functions for half adder

A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

```
Boolean expression equivalents:

Sum = A XOR B
Carry = A AND B
```

Boolean functions for full adder

A	B	C	Sum	Carry
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1

```
Boolean expression equivalents:

Sum = A XOR B XOR C
Carry = (A AND B) OR ((A XOR B) AND C)
```

Binary addition

- The ALU's adder is based on these ideas of half- and full-adders
  - Actual implementations can differ somewhat as the rippling of carries takes time...
  - ... and alternate adder designs reduce or eliminate the latency caused by such latency
  - (But that is a subject for another course.)
- The subtraction hardware is also uses an adder...
  - ... And this is based on turning A – B into A + (-B)

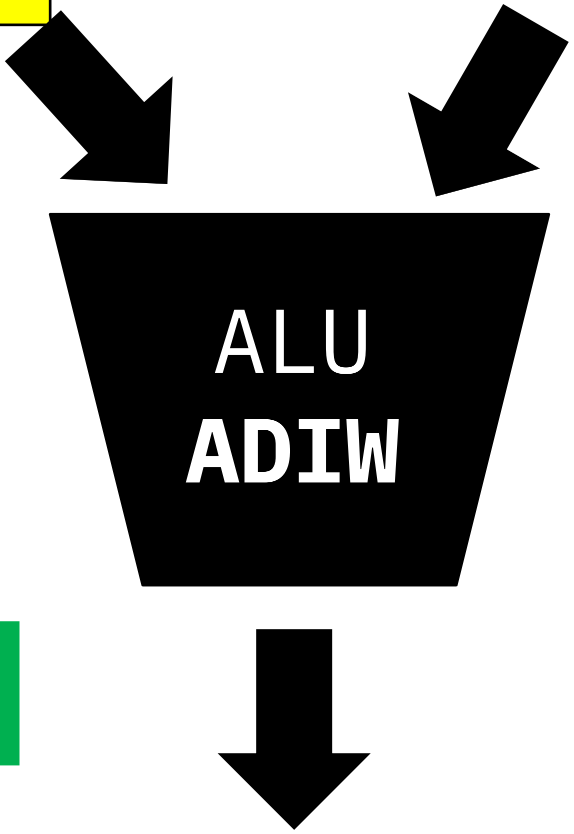
An aside: ADIW

- We will eventually see how the 8-bit addition and subtraction operations help with adding/subtracting larger numbers
- There is, however, some limited support for 16-bit unsigned arithmetic in the instruction set
  - It involves a small immediate (i.e., 0 to 63) ...
  - ... which may be added to pointers / pseudo registers (X, Y, or Z)
  - ... and to which AVR also includes R25:R24
- Motivation: such small additions are common when data in memory (i.e., striding through an array)

Example: ADIW

```
register pair R27:R26
0b00000000:0b11111110
```

```
value of 3
```



All 16-bit address values are, by definition, unsigned

```
ADIW R27:R26, 3
ADIW XH:XL, 3
```

```
result:
R27 = 0b00000001  R26 = 0b00000001
```

Reminder: Untyped data

- Many of the concepts in this lecture may seem (and/or will seem) needlessly complicated.
  - “Isn’t an integer just an integer”
- We are very spoiled by high-level languages
  - Using typed data (i.e., ints, floats, strings, chars, etc.) becomes second nature.
  - The program translator (compiler or interpreter) catches our mistakes when we use data in nonsensical ways.
- In assembler, however, data is not typed!
  - We can treat memory contents as any type we want
  - And that includes making mistakes...
- We ourselves must give data its meaning.

Back to ADD, SUB

- An AVR architecture **does not care** whether we use signed or unsigned integers!
  - This seems confusing at first ...
  - ... but we will see the architecture gives us the tools to use such integers as we intend.
- This fact is important as it helps explain the role of some SREG flags
  - V: **overflow flag**
  - S: **sign-change flag**
  - C: **carry flag**
  - N: **negative flag**
- AVR assembler manual describes what causes these flags to be set (or cleared) by ADD and SUB

Scenario 1

- We want to use one-byte **unsigned** integers
  - That is, bytes will represent positive values in the range of 0 to 255.
  - We are not explicitly concerned here with two's complement numbers.

```
LDI R16, 20      ; 20  = 0b00010100
LDI R18, 120     ; 120 = 0b01111000
ADD R18, R16     ; result is 140 = 0b10001100
```

```
Flags set by this ADD operation: V     N
```

Scenario 1

- We want to use one-byte **unsigned** integers
  - That is, bytes will represent positive values in the range of 0 to 255.
  - We are not explicitly concerned here with two's complement numbers.

```
LDI R16, 140     ; 140 = 0b10001100
LDI R18, 131     ; 131 = 0b10000011
ADD R18, R16     ; result is 15 = 0b00001111
```

```
Flags set by this ADD operation: V S C
```

```
256 + 15 = ?
```

Scenario 2

- We want to use one-byte **signed** integers
  - That is, bytes will represent positive values in the range of 0 to 255.
  - We are not explicitly concerned here with two's complement numbers.

```
LDI R16, 20       ; 20  = 0b00010100
LDI R18, 120      ; 120 = 0b01111000
ADD R18, R16      ; result is -116 = 0b10001100
```

Flags set by this ADD operation: **V** **N**

Scenario 2

- We want to use one-byte **signed** integers
  - That is, bytes will represent positive values in the range of 0 to 255.
  - We are not explicitly concerned here with two's complement numbers.

```
LDI R16, -116    ; -116 = 0b10001100
LDI R18, -125    ; -125 = 0b10000011
ADD R18, R16     ; result is 31 = 0b00001111
```

```
Flags set by this ADD operation: V S C
```

Comparing scenarios...

```
; Scenario 1A
LDI R16, 20      ; 20  = 0b00010100
LDI R18, 120     ; 120 = 0b01111000
ADD R18, R16     ; result is 140 = 0b10001100
```

V N

```
; Scenario 1B
LDI R16, 140     ; 140 = 0b10001100
LDI R18, 131     ; 131 = 0b10000011
ADD R18, R16     ; result is 15 = 0b00001111
```

V S C

```
; Scenario 2A
LDI R16, 20      ; 20  = 0b00010100
LDI R18, 120     ; 120 = 0b01111000
ADD R18, R16     ; result is -116 = 0b10001100
```

V N

```
; Scenario 2B
LDI R16, -116    ; -116 = 0b10001100
LDI R18, -125    ; -125 = 0b10000011
ADD R18, R16     ; result is 31 = 0b00001111
```

V S C

From scenario 1A to 2A, and from 1B to 2B, there was absolutely no change in bit sequences or in flags set. All that changed was our own interpretation!

This can feel confusing...

- ... but only if we are unsure **where** the meaning of numbers is defined
  - We may think positive and negative are meanings "known" by the CPU...
  - ... but really **they are meanings we ourselves make of the bit sequences** stored by the CPU.
- Keeping this clear will help with debugging.
  - Also it helps to be consistent in problem solving
  - That means choosing unsigned or signed for a solution, and sticking with the choice throughout the solution's code.

Two-byte integers

- Values from 0 to 255 (or -128 to 127) can be rather limiting.
- Normally we want a much greater range.
  - two bytes: 0 to 65535 (or -32768 to 32767)
  - four bytes: 0 to 2^{32}-1 (or -2^{31} to 2^{31}-1)
  - eight bytes: 0 to 2^{64}-1 (or -2^{63} to 2^{63}-1)
- If we want a greater range...
  - ... then we must "roll our own"
  - which mean writing the code to add (or subtract) more than two pairs of bytes together.
- Note: Compilers usually do this for us in higher-level languages

Handling the carry

- Recall this scenario:

```
LDI R16, 20       ; 140 = 0b10001100
LDI R18, 120      ; 131 = 0b10000011
ADD R18, R16      ; result is 15 = 0b00001111
```

```
Flags set by this ADD operation: V S C
```

- It may be that we do want to add 140 and 131 as unsigned numbers...
  - ... which means the answer should be 271, or 0x10f
  - This requires two bytes to represent

ADD vs. ADC

- Recall earlier the distinction between half-adder and full-adder
  - The former ignore any incoming carry
  - The latter is able to use an incoming carry
- ADC: “Add with Carry”
  - The two registers provided as arguments are added...
  - **... and the value 1 is also added if the carry flag isset before ADC starts.**

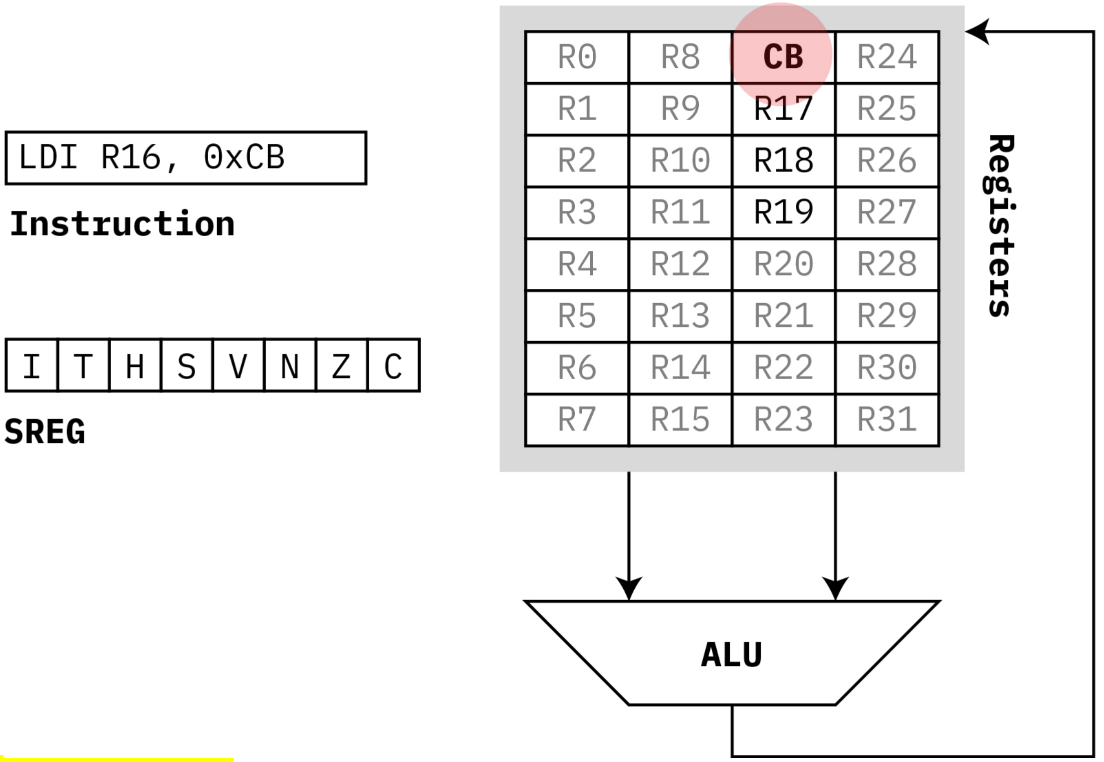
Scenario 3

- We want to use two-byte **unsigned** integers
  - That is, bytes will represent positive values in the range of 0 to 65535.
  - Again we are not explicitly concerned here with two's complement numbers.

```
; decimal 4811 = 0b00010010 11001011
; decimal 9549 = 0b00100101 01001101

LDI R16, 0xCB   ; low byte of 4811 (decimal 203)
LDI R17, 0x12   ; high byte of 4811 (decimal 18)
LDI R18, 0x4D   ; low byte of 9549 (decimal 77)
LDI R19, 0x25   ; high byte of 9549 (decimal 37)
ADD R18, R16    ; add the low bytes together
ADC R19, R17    ; add the high bytes together with possible carry
; result in R19:R18 is 0b00111000 00011000 (0x3818, decimal 14360)
```

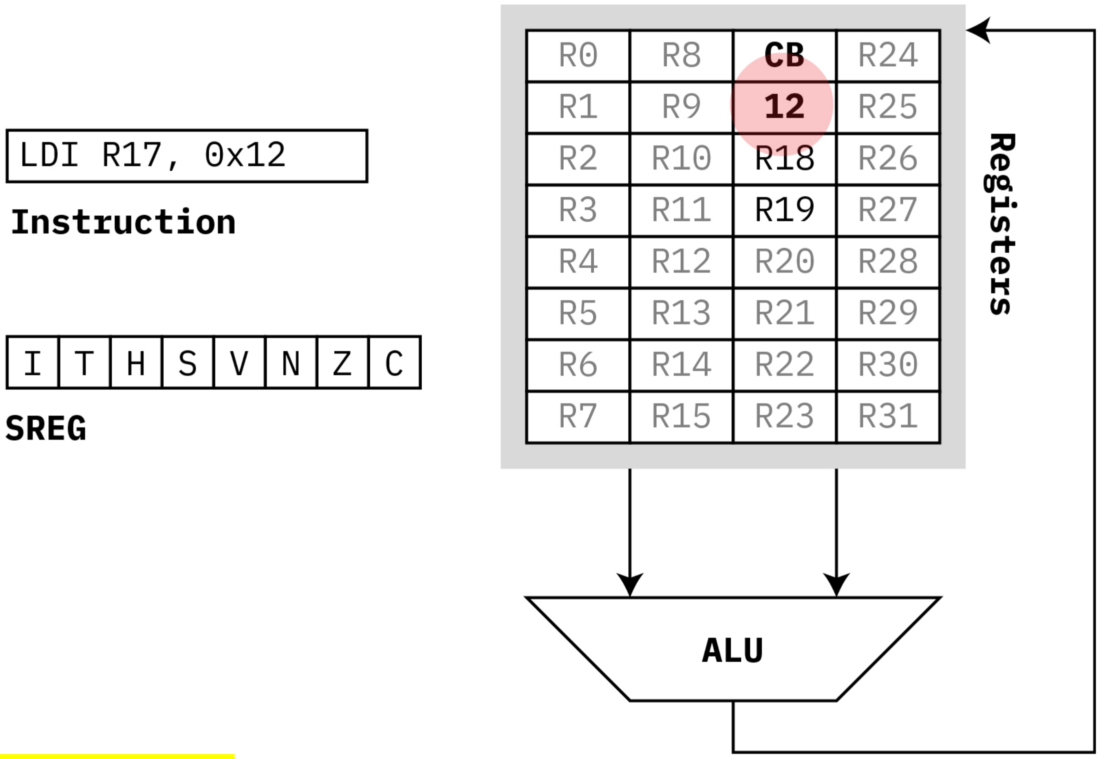
Simulation



```
LDI R16, 0xCB
LDI R17, 0x12
LDI R18, 0x4D
LDI R19, 0x25
ADD R18, R16
ADC R19, R17
```

Loading decimal 4811 into R17:R16

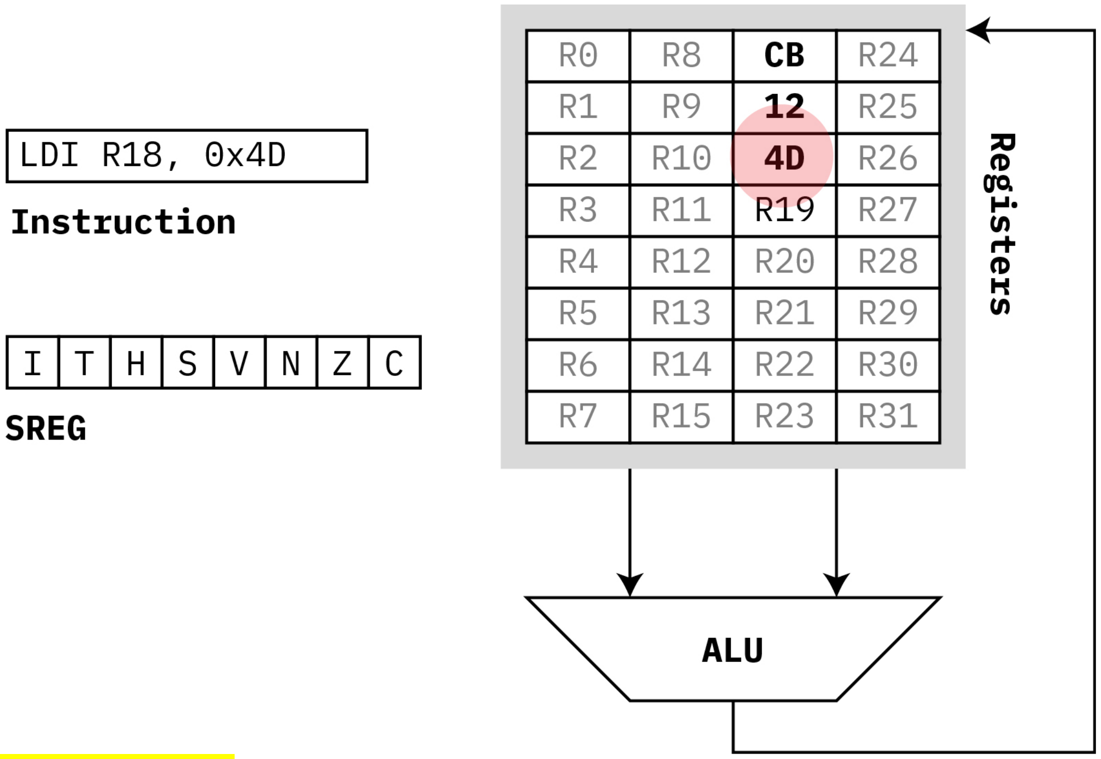
Simulation



Loading decimal 4811 into R17:R16

```
LDI R16, 0xCB
LDI R17, 0x12
LDI R18, 0x4D
LDI R19, 0x25
ADD R18, R16
ADC R19, R17
```

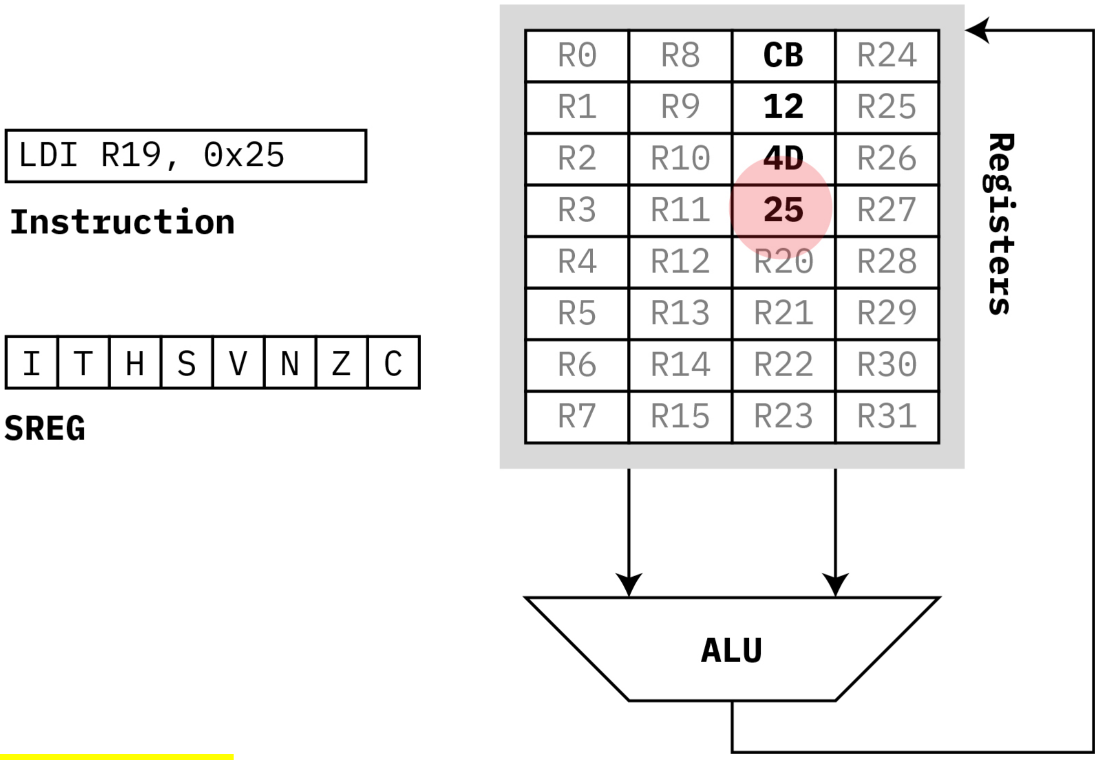
Simulation



```
LDI R16, 0xCB
LDI R17, 0x12
LDI R18, 0x4D
LDI R19, 0x25
ADD R18, R16
ADC R19, R17
```

Loading decimal 9549 into R19:R18

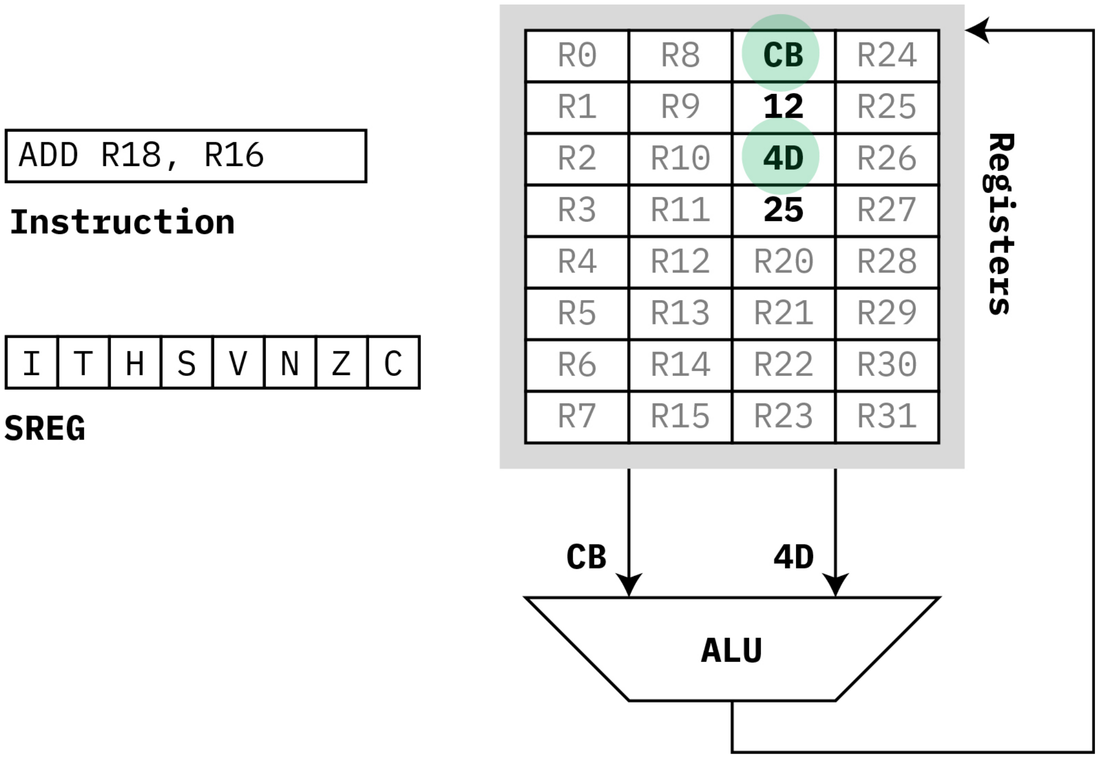
Simulation



Loading decimal 9549
into R19:R18

```
LDI R16, 0xCB
LDI R17, 0x12
LDI R18, 0x4D
LDI R19, 0x25
ADD R18, R16
ADC R19, R17
```

Simulation

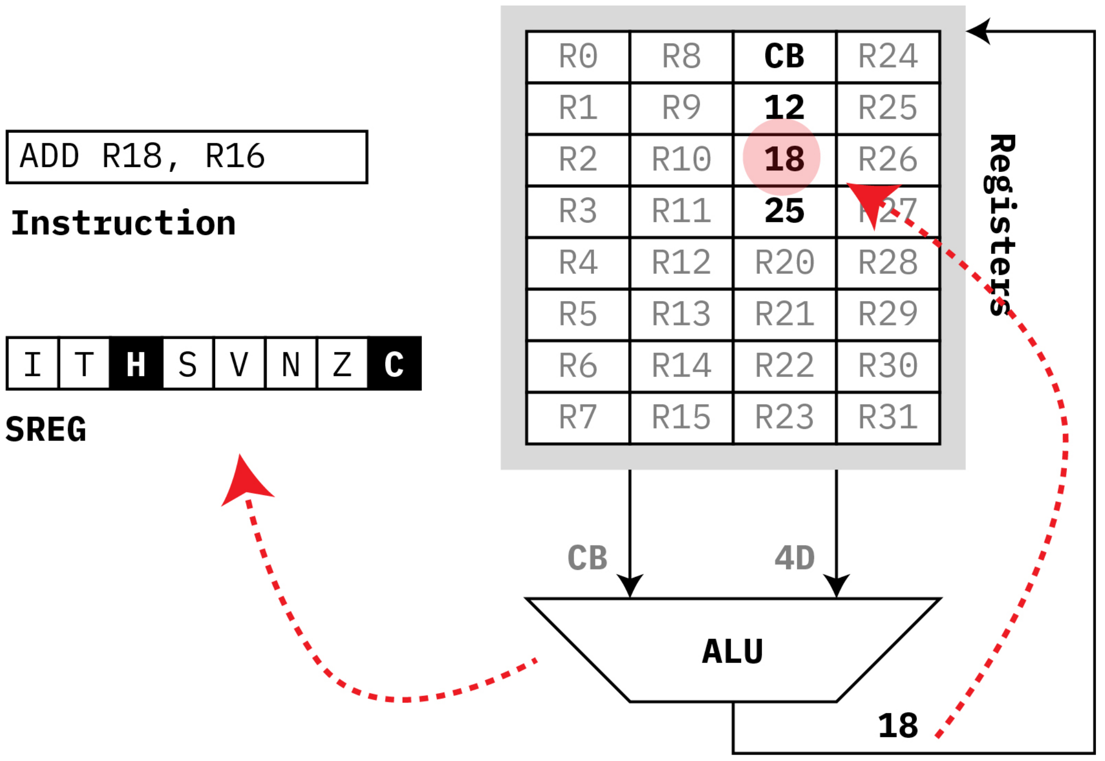


Add low bytes

(first part of execution)

```
LDI R16, 0xCB
LDI R17, 0x12
LDI R18, 0x4D
LDI R19, 0x25
ADD R18, R16
ADC R19, R17
```

Simulation

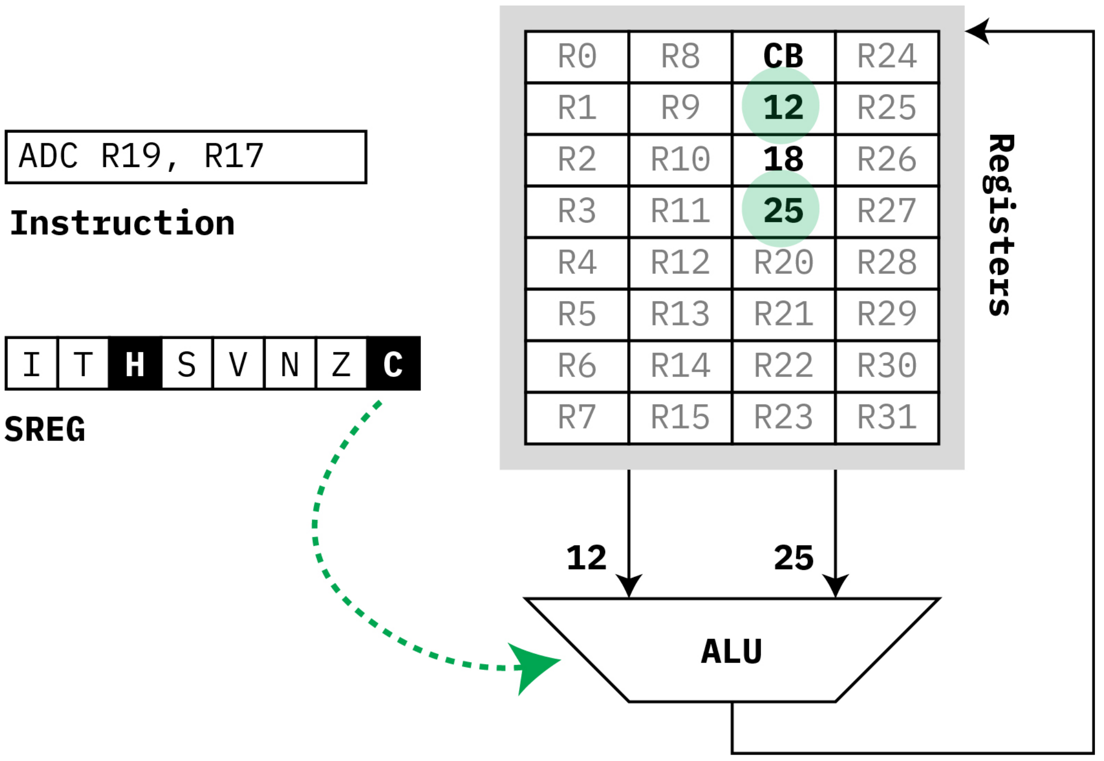


Add low bytes

(second part of execution)

```
LDI R16, 0xCB
LDI R17, 0x12
LDI R18, 0x4D
LDI R19, 0x25
ADD R18, R16
ADC R19, R17
```

Simulation

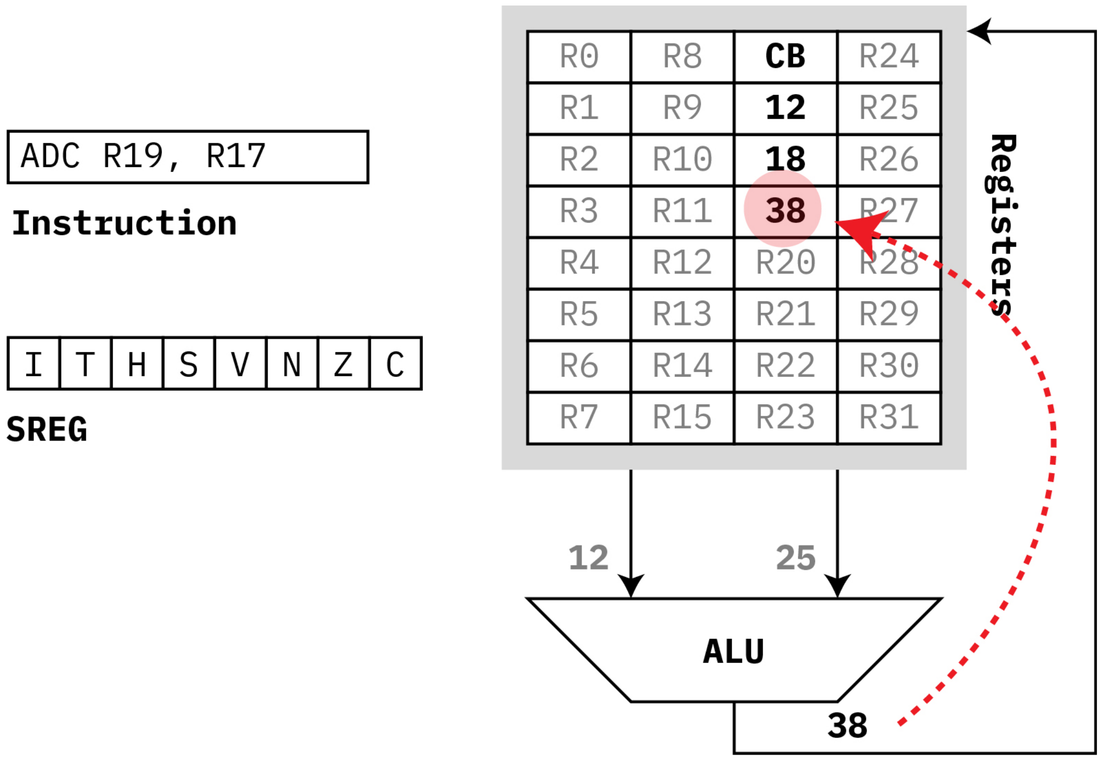


Add high bytes

(first part of execution)

```
LDI R16, 0xCB
LDI R17, 0x12
LDI R18, 0x4D
LDI R19, 0x25
ADD R18, R16
ADC R19, R17
```

Simulation



Add high bytes

(second part of execution)

```
LDI R16, 0xCB
LDI R17, 0x12
LDI R18, 0x4D
LDI R19, 0x25
ADD R18, R16
ADC R19, R17
```

SUB vs. SBC

- Something similar is available with subtraction
  - SUB: Subtract
  - SBC: Subtract with carry
- However, the “carry” actually behaves more like a “borrow”
  - With addition we go from low-byte to high-byte…
  - … and we follow the same pattern with subtraction.
- Note, though, that we need not do any two’s complement conversion
  - The ALU will do this for us.

Scenario 4

- We want to use two-byte **signed** integers
  - That is, bytes will represent positive values in the range of -32768 to 32767.
- The meaning of high-byte and low-byte do not change
  - We'll interpret the result as a 16-bit two's complement number

```
; decimal 4811 = 0b00010010 11001011
; decimal 9549 = 0b00100101 01001101
; compute: 4811 - 9549

LDI R16, 0xCB   ; low byte of 4811 (decimal 203)
LDI R17, 0x12   ; high byte of 4811 (decimal 18)
LDI R18, 0x4D   ; low byte of 9549 (decimal 77)
LDI R19, 0x25   ; high byte of 9549 (decimal 37)
SUB R16, R18    ; subtract the low bytes; notice order of operands
SBC R17, R19    ; subtract high bytes with possible borrow
; result in R17:R16 is 0b11101101 01111110 (0xED7E, decimal -4738)
```

15 – 20 = ?

- During subtraction of unsigned numbers:
  - It is possible the **subtrahend** (i.e., 20 in the example above)...
  - .... is larger than the **minuend** (i.e., 15 in the example).
- Given the operands are unsigned...
  - ... it make no sense to refer to a negative number, but rather of an underflow.
- If these operands were the low bytes of a 16-bit unsigned number...
  - ... then a further subtraction would know that a borrow was needed.
  - The AVR architecture indicates a needed borrow by indicating a carry!

Scenario 5

- Store decimal 70 into a register...
- ... and repeatedly subtract 30 from that register
- We will let the assembler convert decimal values into the bit-sequence equivalents for us.

```
LDI R17, 70
LDI R16, 30
SUB R17, R16
SUB R17, R16
SUB R17, R16
```

```
(After the instruction on left...)
R17 value: 0x46  Flags set:
R17 value: 0x28  Flags set: H
R17 value: 0x0A  Flags set: H
R17 value: 0xEC  Flags set: H S N C
```

Scenario 6

- Similar to Scenario 5
- Set R17 to 221
  - Then repeatedly subtract 30 from R17...
  - ... but use a loop (i.e., continue loop as long as the carry flag is not set)

```
    LDI R17, 221
    LDI R16, 30

LOOP:
    SUB R17, R16
    BRCC LOOP      ; Branch if Carry Cleared

STOP:
    RJMP STOP
```

Large variety of branch ops

- The previous example hints at why there are so many different branch instructions
- We can use the flags set (or cleared) in the status register...
  - ... as a way of determine control flow for our program.
- This therefore also means we are in charge of the interpretation of memory contents
  - If we want to treat bit-sequences as unsigned, we can.
  - If we want to treat bit-sequences as signed, we can.
- Later in the course we will look at even more support for other number formats (floating point, BCD)

Larger integers

- In most programming languages, a 16-bit integer is called a short integer
- When using integers in Java, Python or C, we expect 32-bit integers
  - That is, an integer has four bytes
  - In AVR we would call such an amount of memory a double-word
- Adding and subtracting double-words is simply more of the same
  - Usual practice is to load bytes into registers
  - (That is, looping code is rarely written.)

32-bit integers (addition)

```
; decimal 1073743811 = 0x400007C3 (this will be M)
; decimal 535725793 = 0x1FEE86E1 (this will be N)
; compute: M + N

.equ M=1073743811
.equ N=535725793

LDI R16, (M & 0xff)
LDI R17, ((M >> 8) & 0xff)
LDI R18, ((M >> 16) & 0xff)
LDI R19, ((M >> 24) & 0xff)
LDI R20, (N & 0xff)

LDI R21, ((N >> 8) & 0xff)
LDI R22, ((N >> 16) & 0xff)
LDI R23, ((N >> 24) & 0xff)

ADD R20, R16
ADC R21, R17
ADC R22, R18
ADC R23, R19
```

`Result in R23:R22:R21:R20 is 0x5FEE8Ea4`

Summary

- We have looked at addition and subtraction of integers bigger than one byte
- The meaning of the contents in those bytes depends upon our interpretation of SREG flags
  - signed ...
  - ... or unsigned ...
  - depends upon branches we take.
- Straightforward to extrapolate to larger numbers (i.e., from 4 bytes to 8 bytes, etc.).
  - But must ensure we use the correct versions of add and subtract (i.e., with or without carry)