04- AVR Arithmetic

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

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

Outline

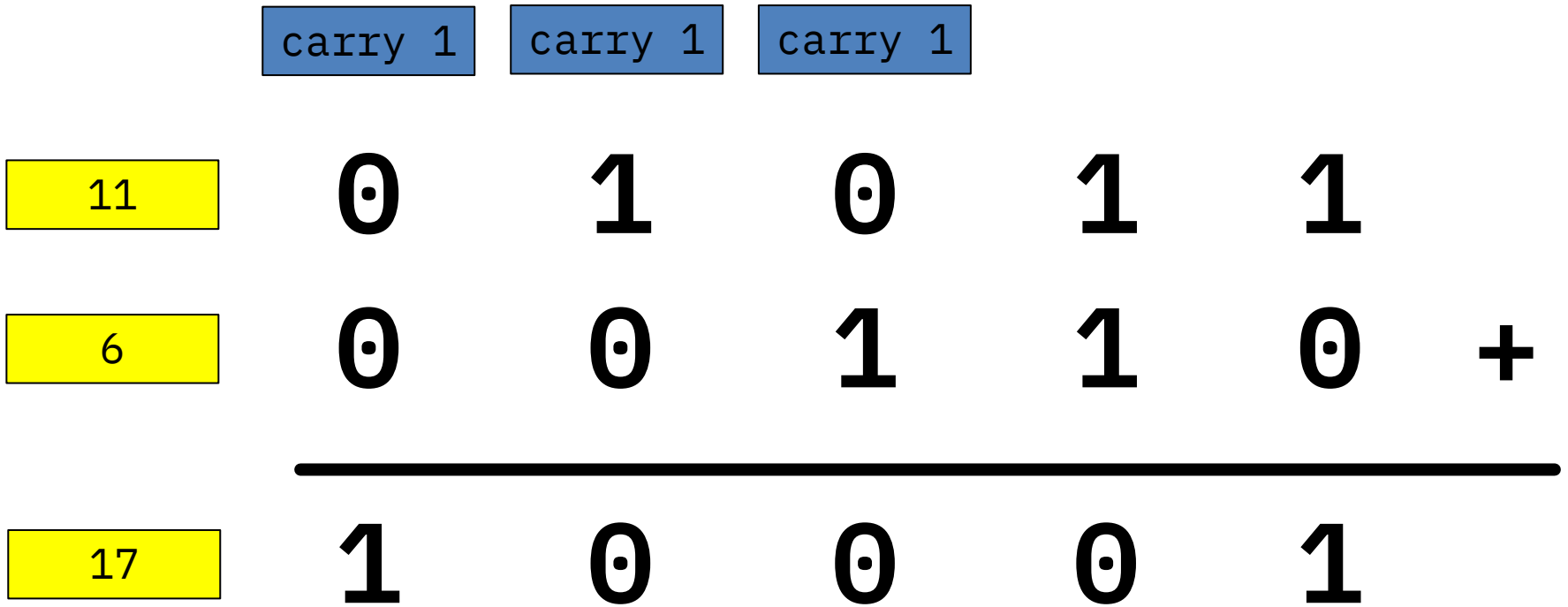
- Relationship between number representations and arithmetic operations
- Aspects of AVR design that permits us to use numbers as we intend
- Adding & subtracting one-byte numbers
- Arithmetic with pointers
- Adding & subtracting two-byte and four-byte numbers

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



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., if result of add is 1)
  - And must 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)

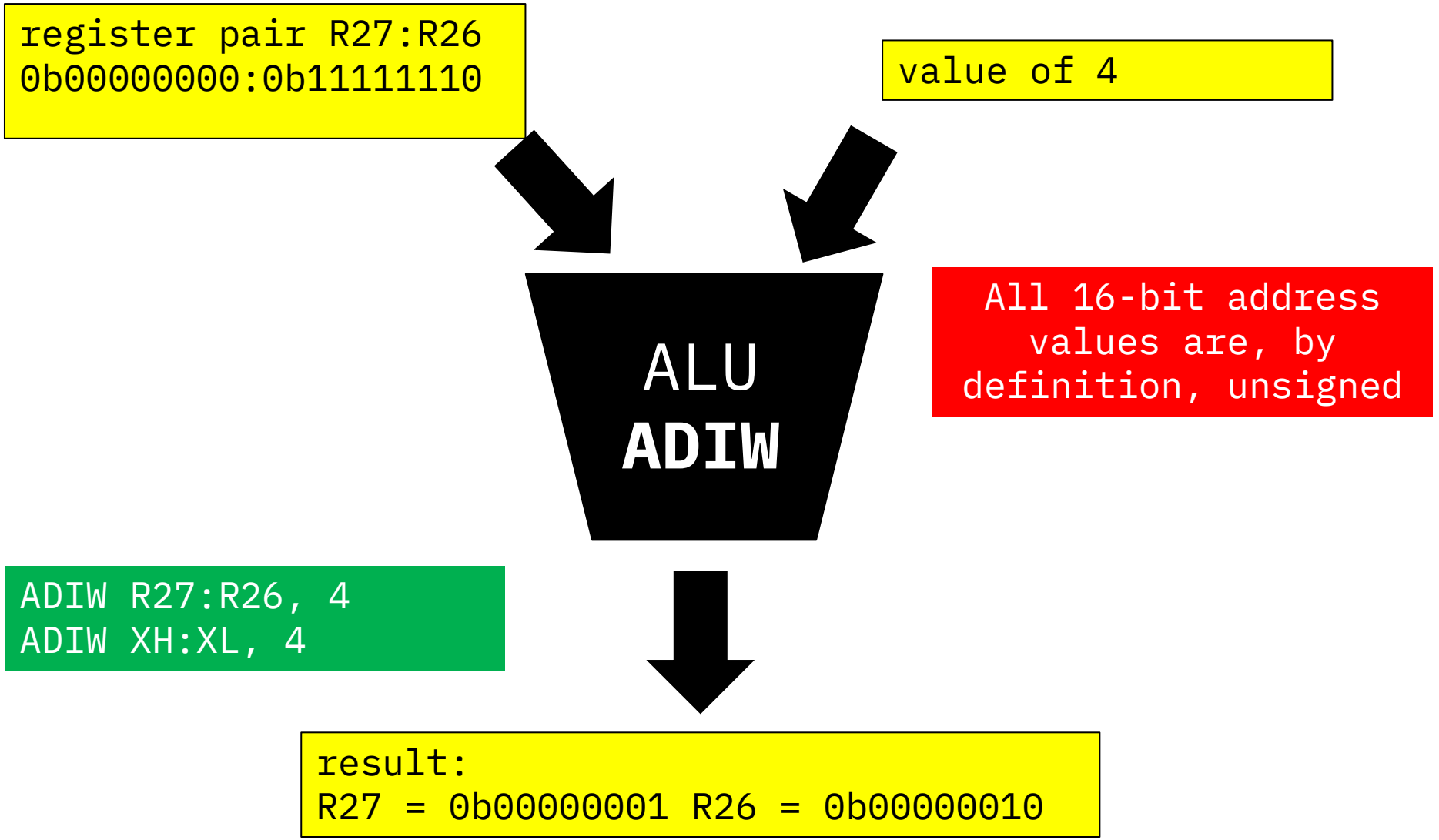
Binary addition

- 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 ripples that cause such delays
  - (But that is a subject for another course.)
- The subtraction hardware 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 can be combined to add/subtract numbers needing more than 8-bits
- 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 accessing data in memory (i.e., striding through an array)

Example: ADIW



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? C’mon!!”
- 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: two's complement 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 (amongst other instructions).

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 (cont.)

- 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
```

```
255 + 15 = ?
```

Scenario 2

- We want to use one-byte signed integers
  - That is, bytes will represent values in the range of -128 to 127.

```
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 (cont.)

- We want to use one-byte signed integers
  - That is, bytes will represent values in the range of -128 to 127.

```
LDI R16, -116   ; -116 = 0b10001100
LDI R18, -125   ; -125 = 0b10000011
ADD R18, R16    ; result is 15 = 0b00001111
```

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

```
Note: -241 in 9-bit 2's complement is represented
as 0x10F
```

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 15 = 0b00001111
```

-241 = 0x10f

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 your solution’s code.

Any Questions?