CS1021 Introduction to Computing I
3. Arithmetic

Rebekah Clarke
clarker7@scss.tcd.ie

Binary Arithmetic – Addition

Operates as with decimal addition except that you carry a 2 (10_2) rather than a ten (10_{10}) i.e. $1_2+1_2 = 10_2$

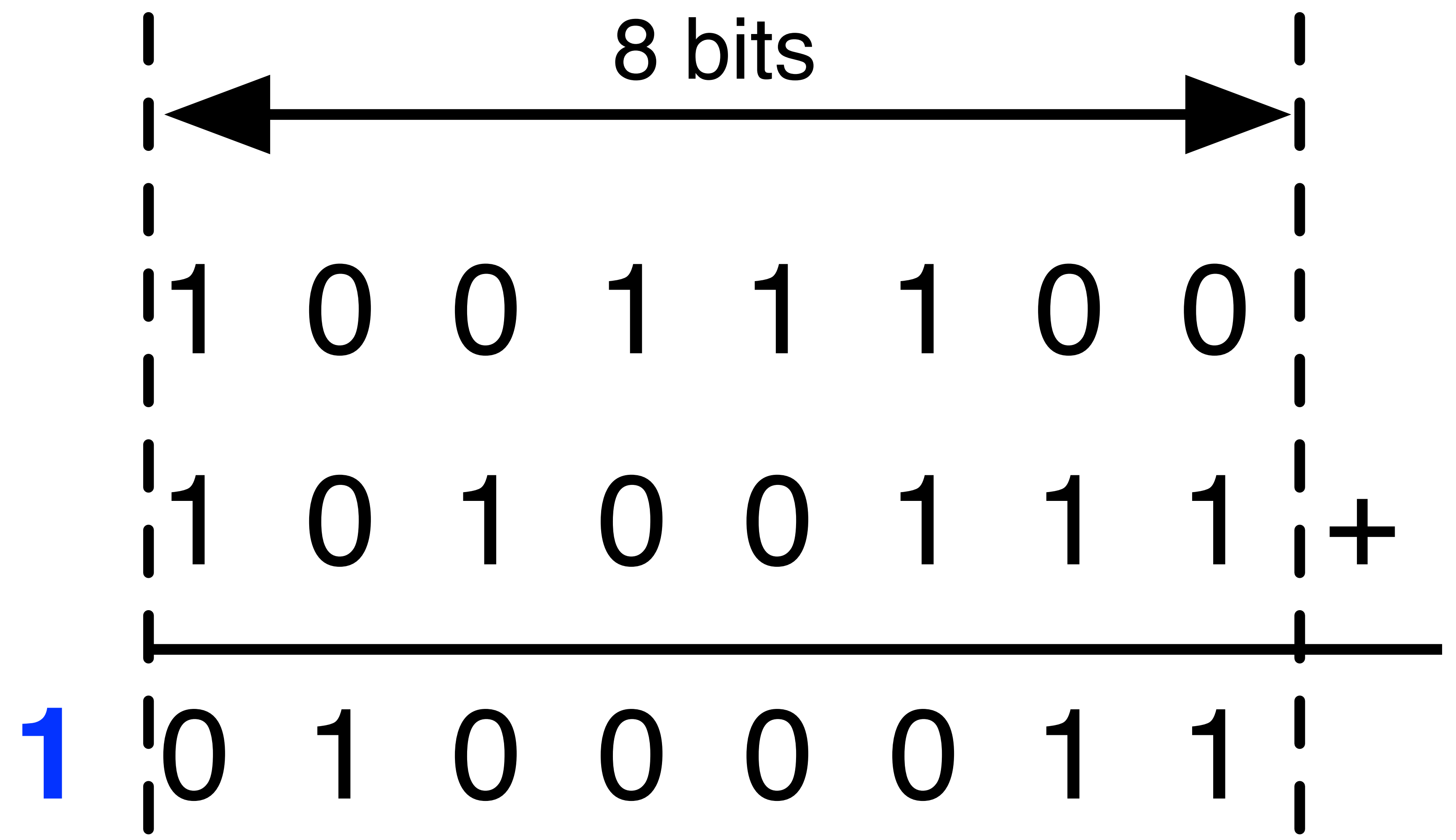
$$\begin{array}{cccccccccc}
0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & & \\
0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & + & \\
\hline
0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & &
\end{array}
\qquad\qquad
\begin{array}{r}
6 \\
11\ + \\
\hline
17
\end{array}$$

$$\begin{array}{cccccccccc}
0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & & \\
0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & + & \\
\hline
0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & &
\end{array}
\qquad\qquad
\begin{array}{r}
22 \\
11\ + \\
\hline
33
\end{array}$$

Carry

What happens if we run out of digits during binary arithmetic?

Adding two numbers each stored in 1 byte (8 bits) may produce a 9-bit result



$$\begin{array}{r} 156 \\ 167\ + \\ \hline 323 \end{array}$$

Added 156_{10} + 167_{10} and expected to get 323_{10}

8-bit result was 01000011_2 or 67_{10}

Largest number we can represent in 8-bits is 255

The “missing” or “left-over” 1 is called a **carry** (or **carry-out**)

Negative Number Representation

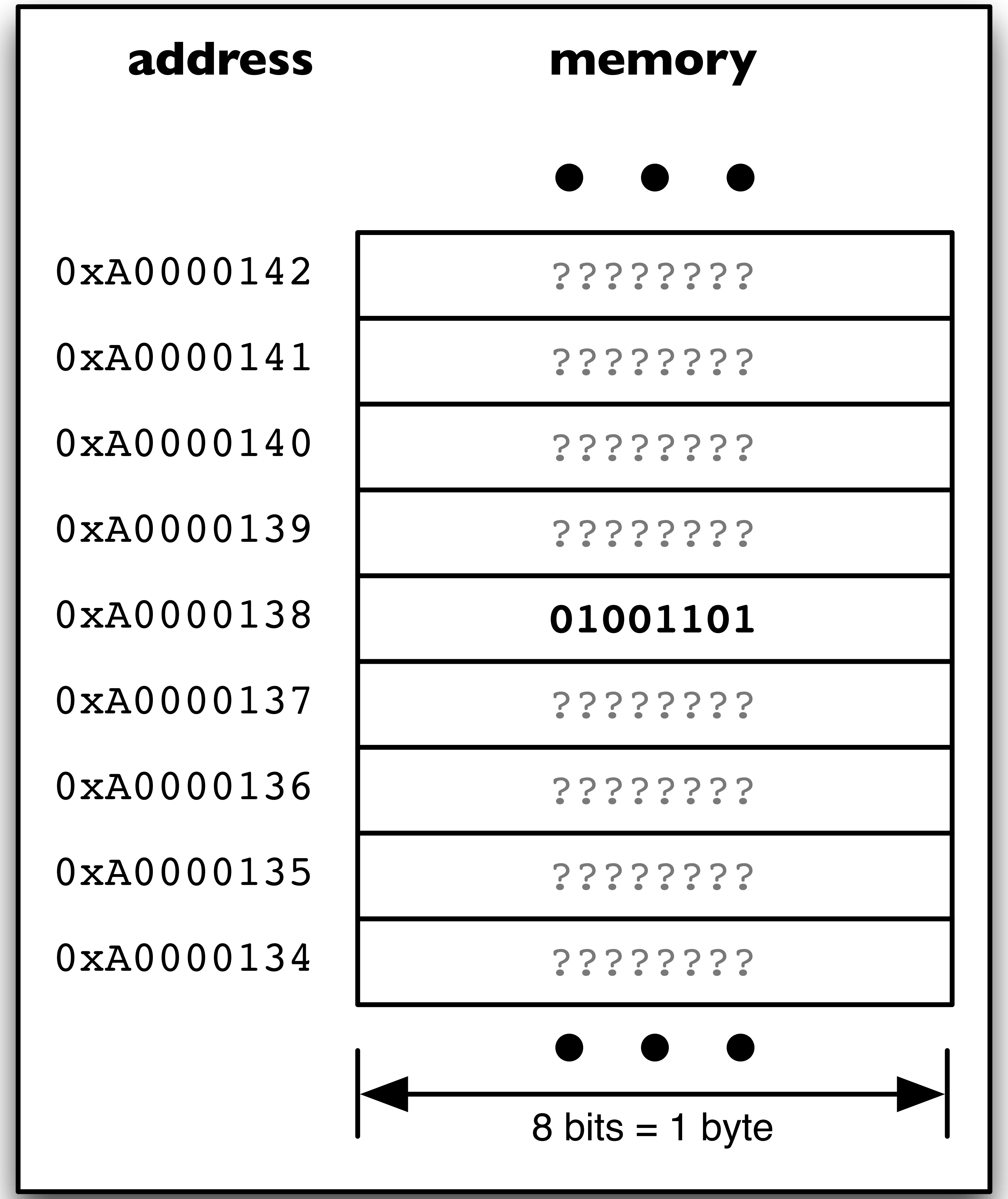
What does the binary value stored in memory at address 0xA0000138 represent?

Depends on **Interpretation!**

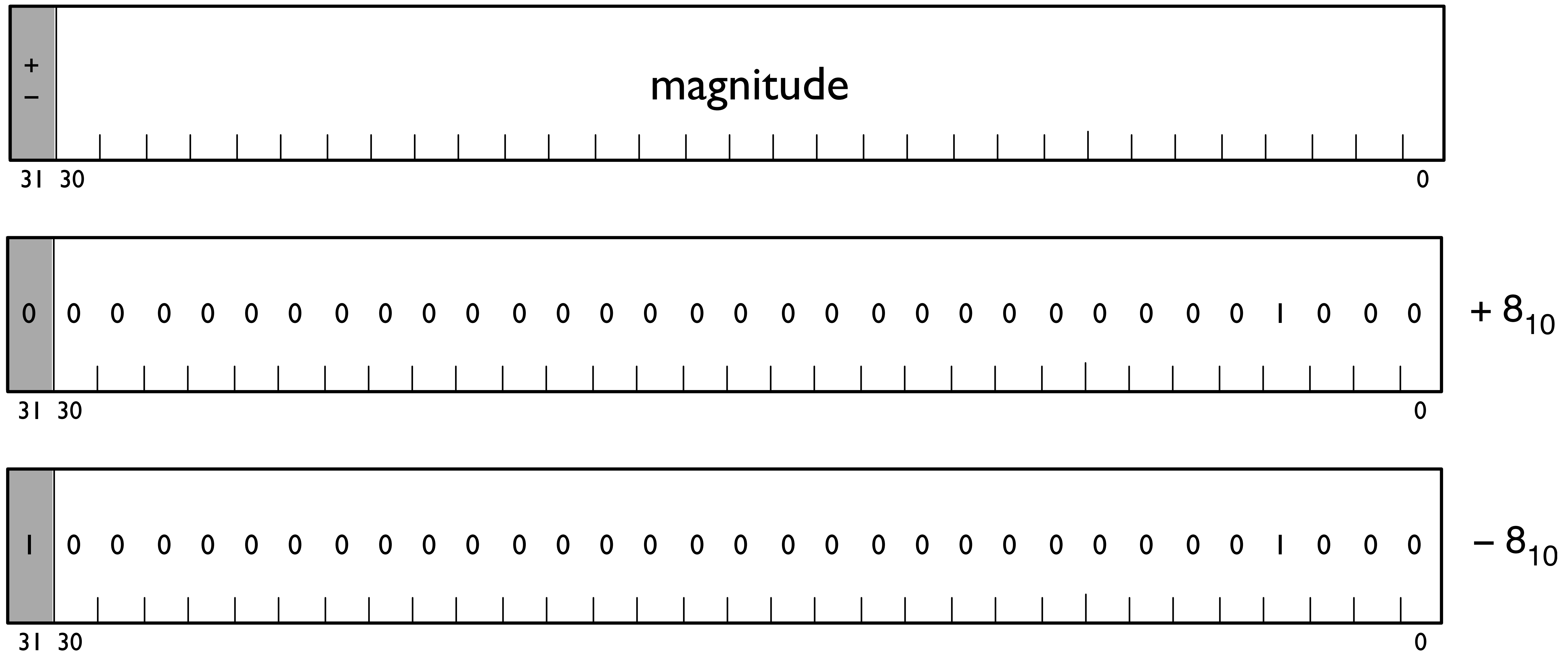
How can we tell whether any given value in memory represents an unsigned value, a signed value, an ASCII character etc?

We can't **tell**, we have to **know**

How can we represent signed values, and negative values such as -17_{10}, in memory?



Sign-Magnitude (this is **not** how we usually represent negative numbers!!)



Can represent signed values in the range [$-2^{n-1}-1$ ... $+2^{n-1}-1$]

e.g. For 32 bits we can represent $-2^{31}-1$ ... $+2^{31}-1$

Two representations of zero (+0 and -0)

Need a special way to handle signed arithmetic (complex)

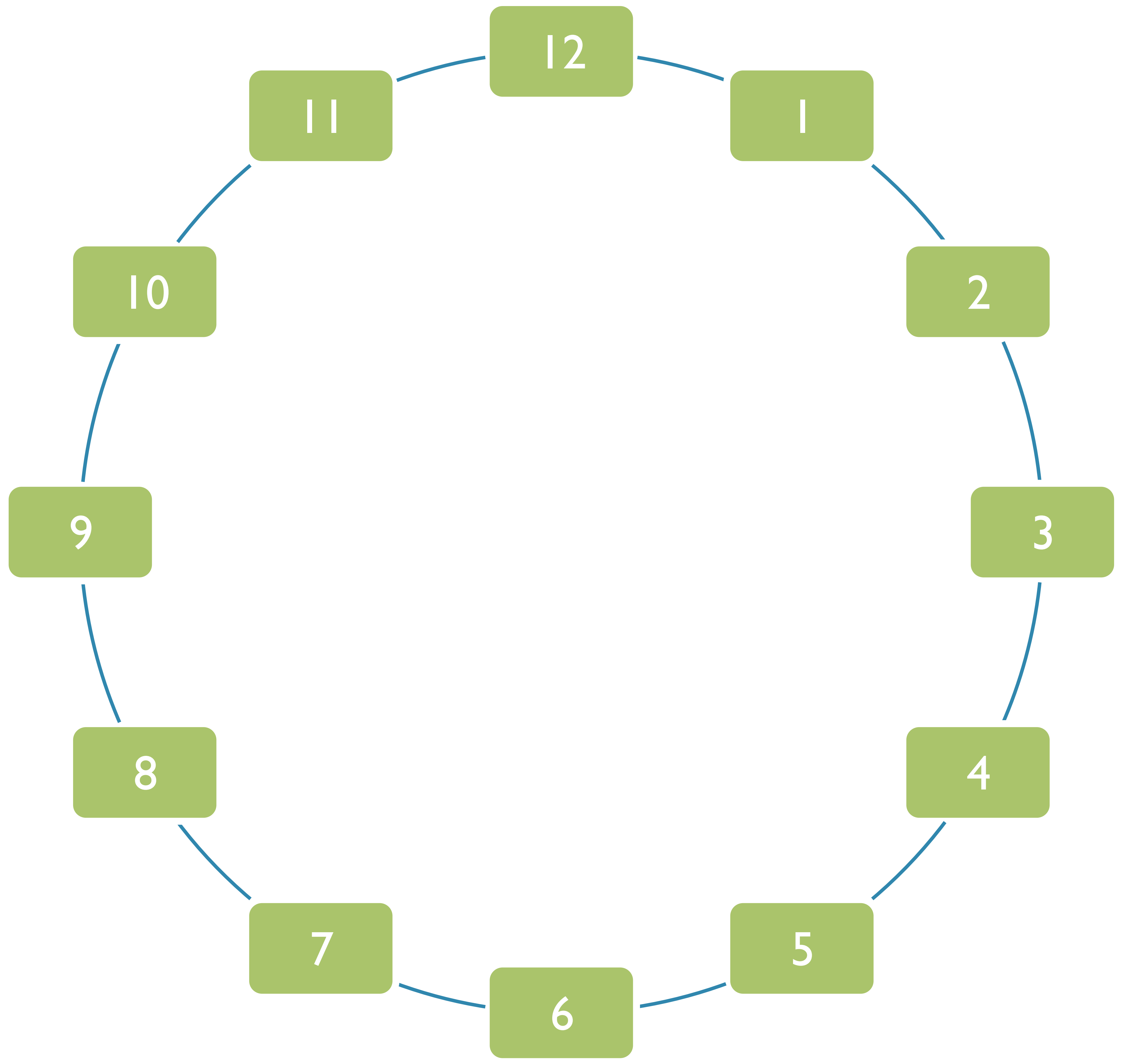
Remember: **interpretation!** (is it -8 or 2,147,483,656?)

Modulo Arithmetic

A 12-hour clock is an example of modulo-12 arithmetic

If we add 4 hours to 10 o'clock we get 2 o'clock

If we subtract 4 from 2 o'clock we get 10 o'clock (not -2 o'clock!)



Two’s Complement

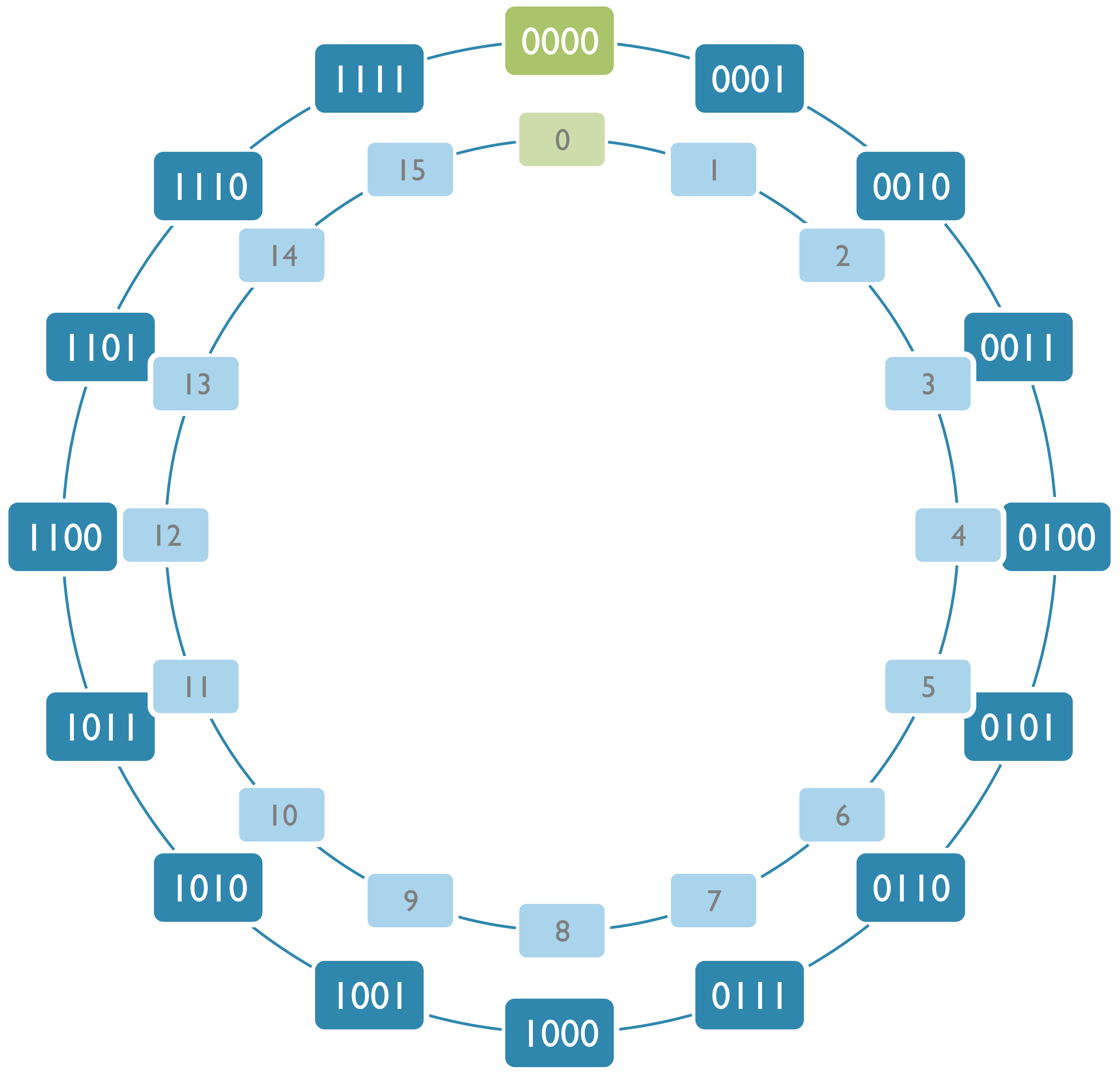
Can represent 16 values with a 4-bit number system ($2^4 = 16$)

Ignoring carries from 4-bit binary addition gives us modulo-16 arithmetic

(15 + 1) mod 16 = 0

(14 + 2) mod 16 = 0

(14 + 4) mod 16 = 2



Two’s Complement

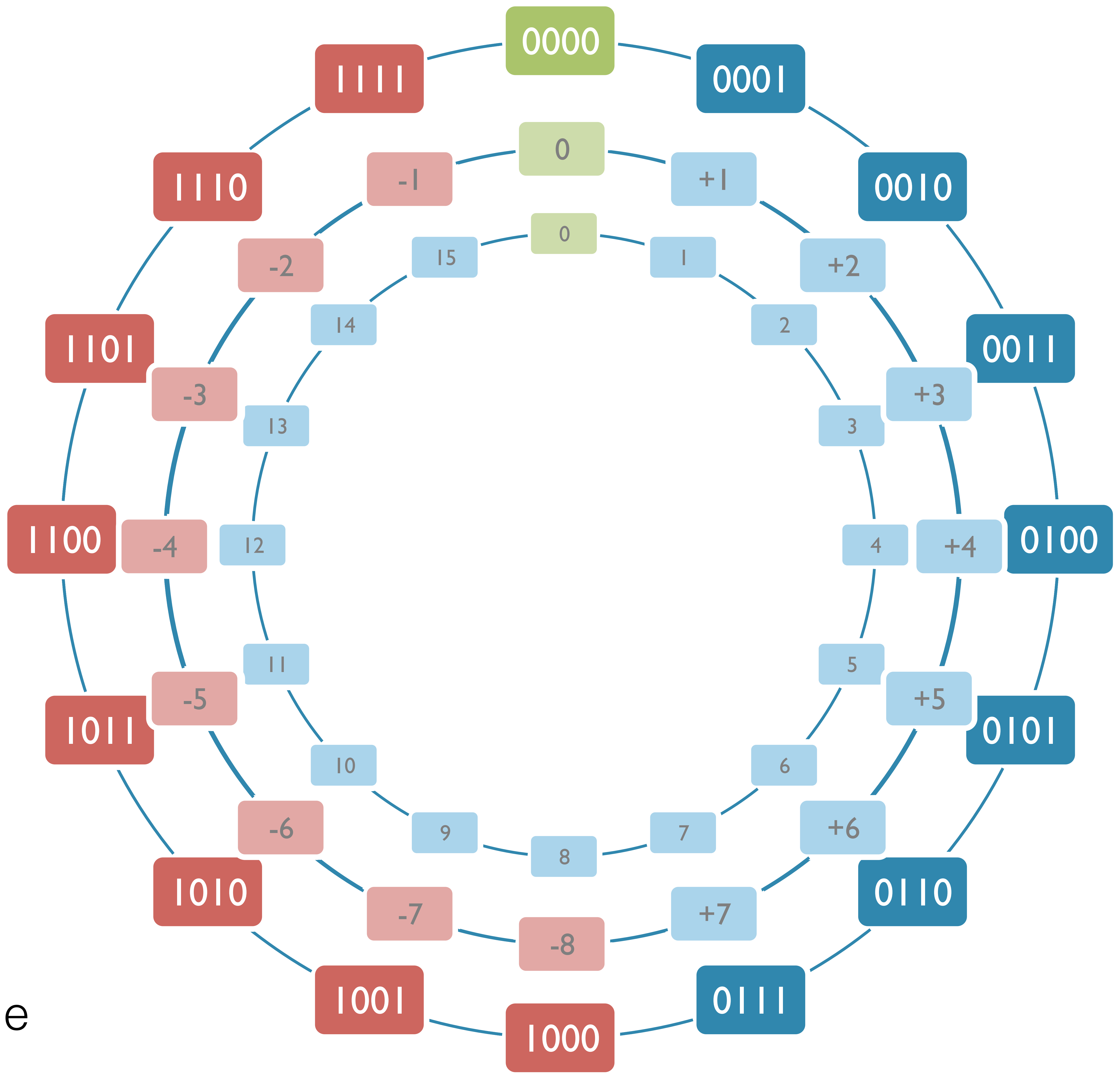
Two's complement is the most common method for representing negative numbers

Utilises modulo arithmetic based on the number of available bits

- e.g. Modulo-16 for 4 bits

An n-bit number's two's complement is the result of subtracting it from 2^n

- Equivalent to inverting the bits of the absolute value and adding one



Two's Complement

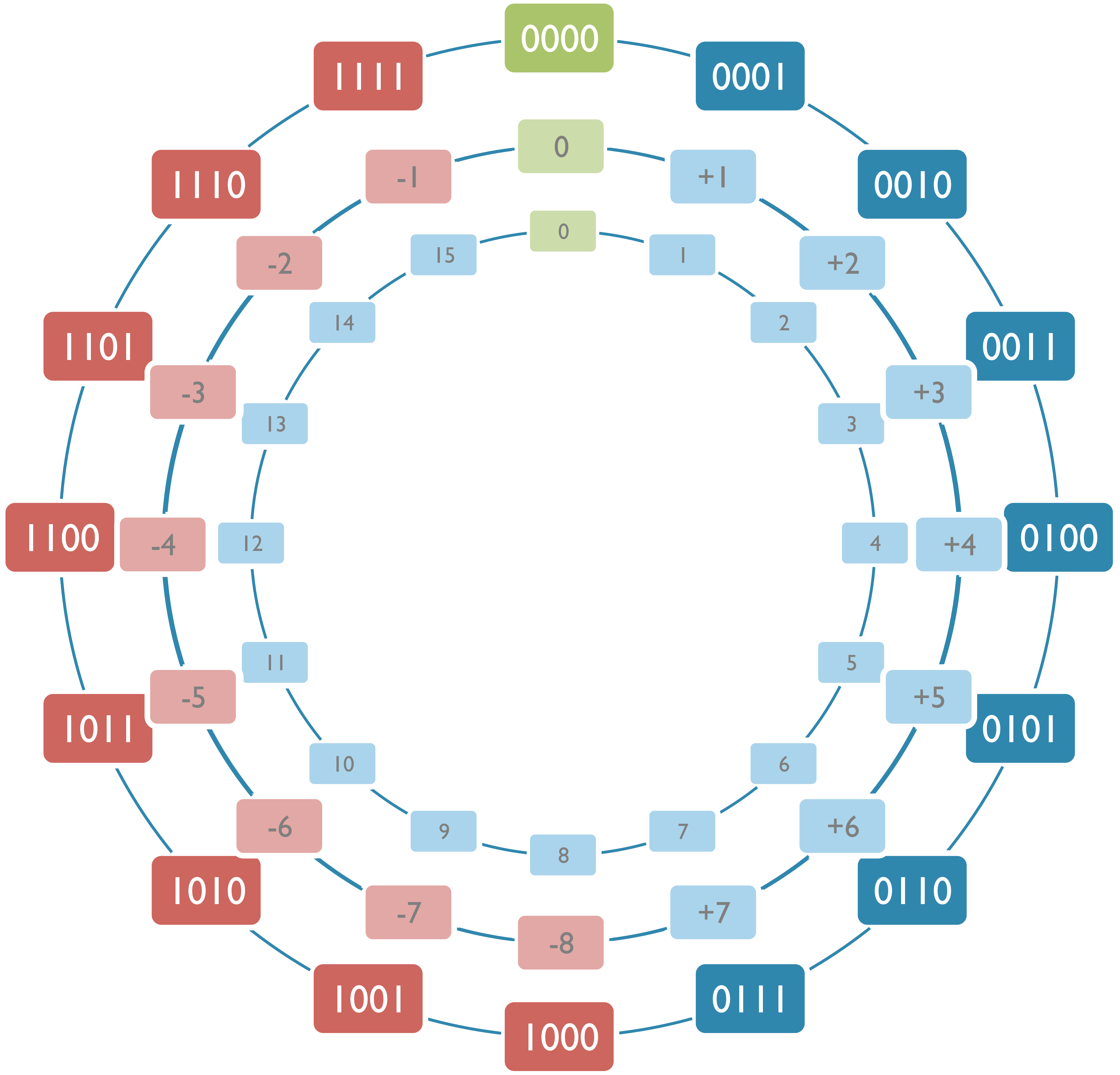
(15 + 1) mod 16 = 0 *and* -1 + 1 = 0

(14 + 2) mod 16 = 0 *and* -2 + 2 = 0

(14 + 4) mod 16 = 2 *and* -2 + 4 = 2

Remember: **Interpretation**

4-bits for comprehension only. Our ARM microprocessor stores 32-bit values and performs 32-bit arithmetic



Two’s Complement

How many representations for zero are there in 2’s Complement?

What is the range of signed values that can be represented with 32 bits using the 2’s Complement system?

How can you tell whether a value represented using a 2’s Complement system is positive or negative?

How can we change the sign of a number represented using a 2’s Complement number system?

How would the values -4 and +103 be represented using a 32-bit 2’s Complement system?

Two’s Complement Examples

8-bits is for illustration only

Represent -97_{10} using Two’s complement

$97_{10} = 01100001_2$

Inverting gives 10011110_2

Adding 1 gives 10011111_2

Interpreted as a Two’s complement signed integer

$10011111_2 = -97_{10}$

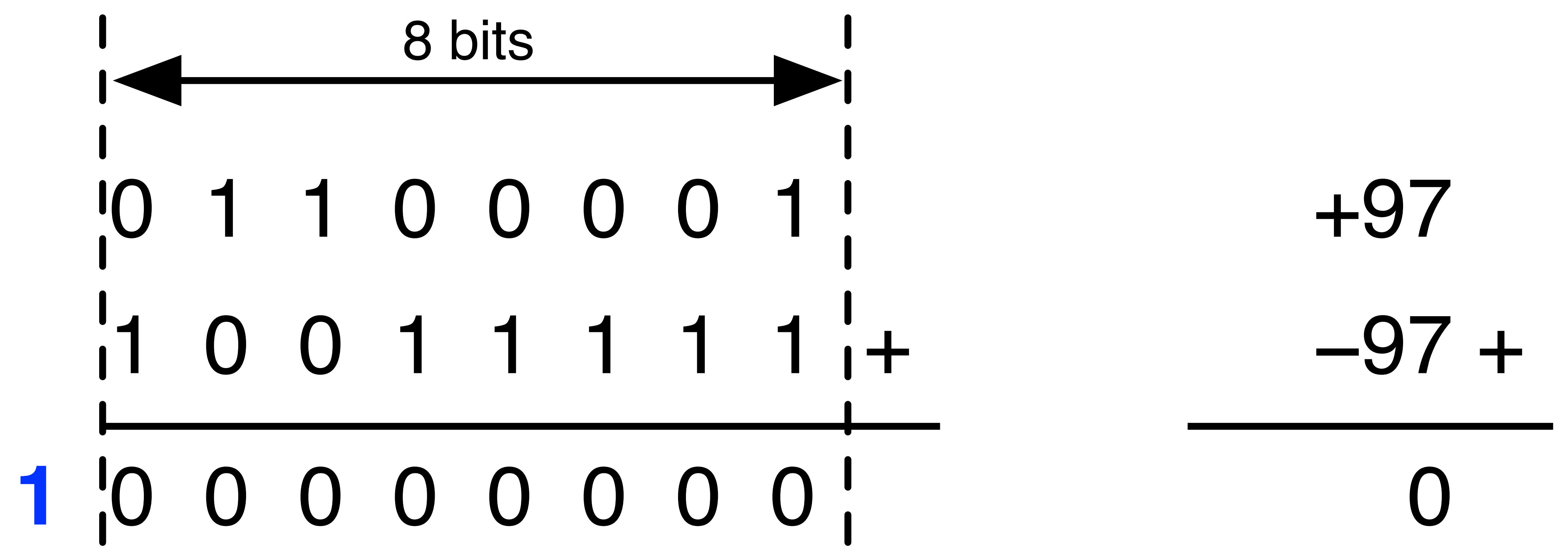
Interpreted as an unsigned integer

$1001\ 1111_2 = 159_{10}$

Correct interpretation is the responsibility of the programmer, not the CPU, which does not “know” whether a value 10011111_2 in R0 is -97_{10} or 159_{10}

Two's Complement Examples

Adding 01100001_2 ($+97_{10}$) and 10011111_2 (-97_{10})



Ignoring the carry bit gives us the correct result of 0

Changing sign of $1001\ 1111_2$ (-97_{10})

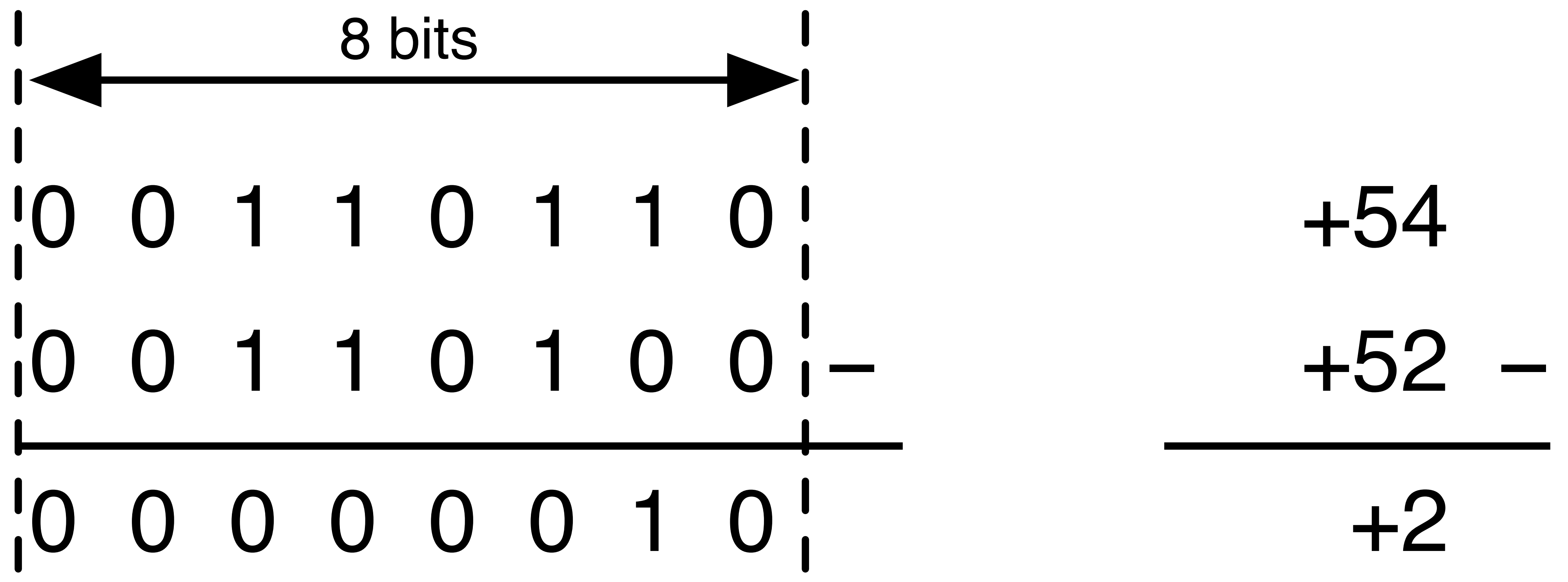
Invert bits and add 1 again

Inverting gives 01100000_2

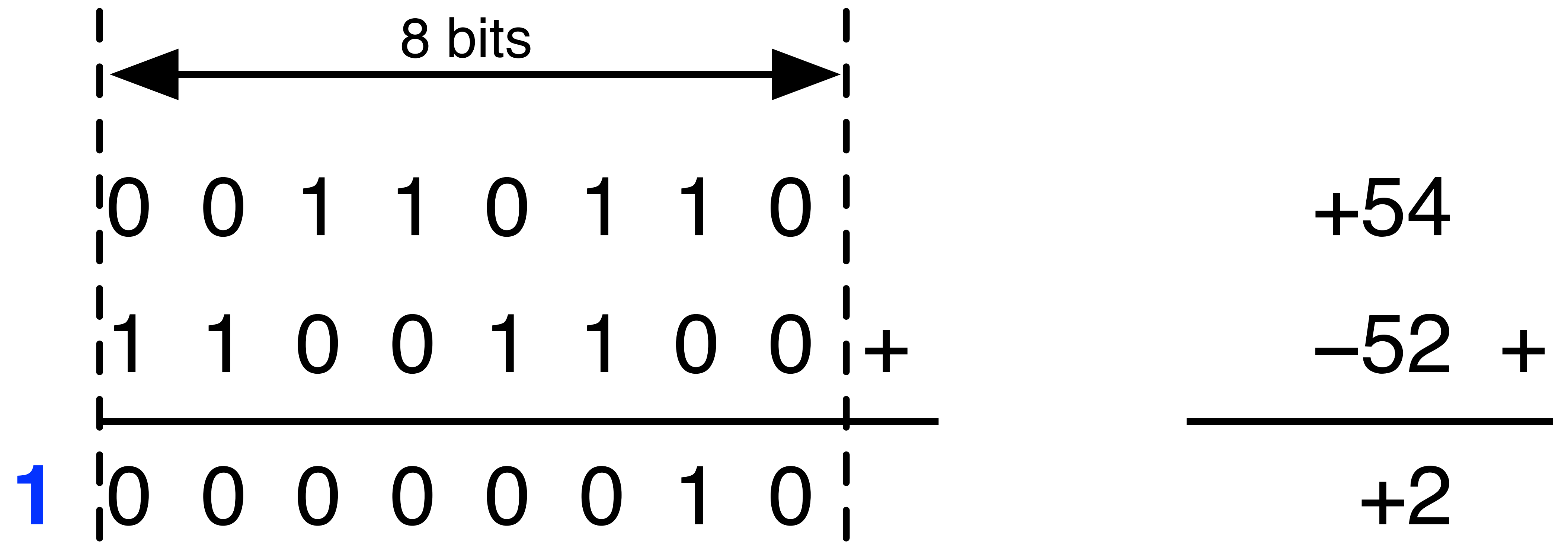
Adding 1 gives 01100001_2 ($+97_{10}$)

Subtraction

A – B

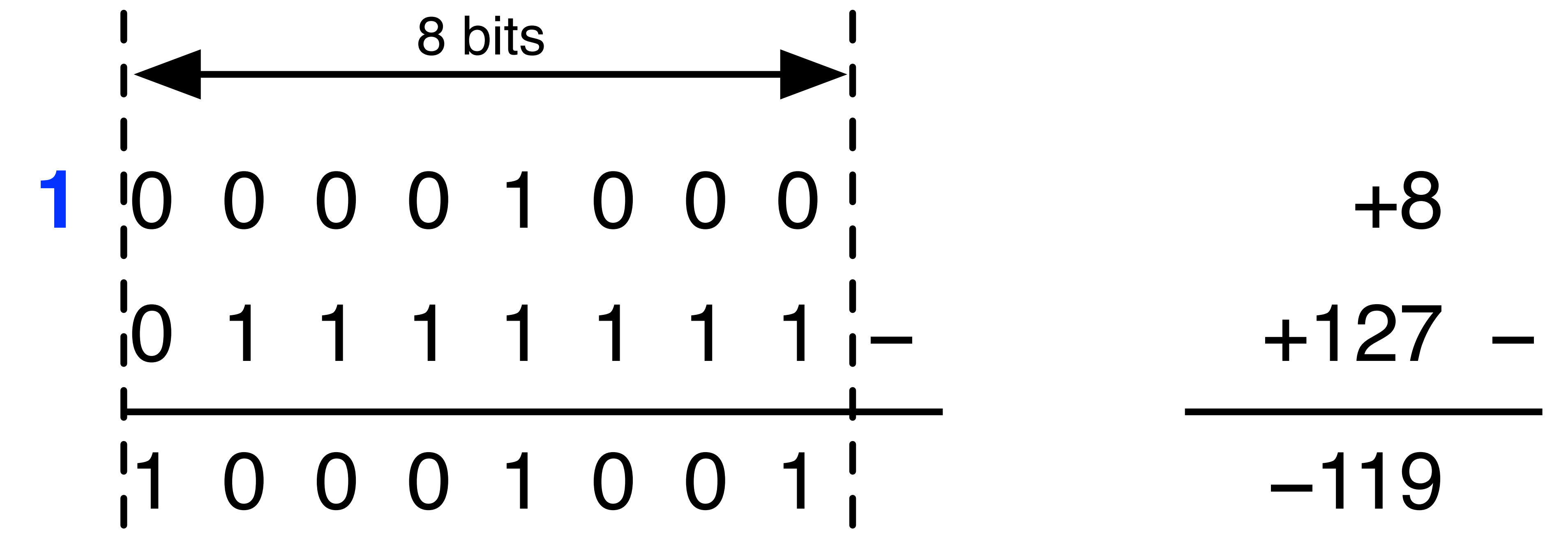


A + TwosComplement(B)

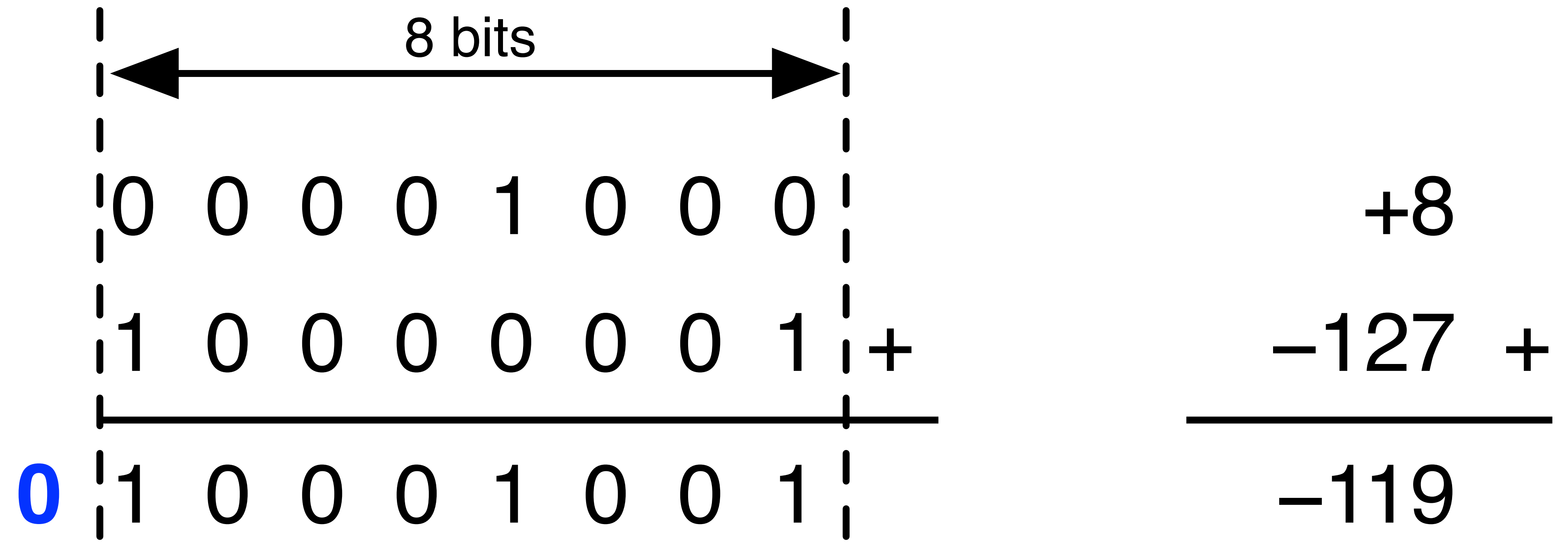


Subtraction

A – B



A + TwosComplement(B)



Changing Sign in Assembly Langauge

Write an Assembly Language program to change the sign of the value stored in R0

Sign of a 2's Complement value can be changed by inverting the value (bits) and adding 1

```
start
        LDR   r0, =7                ; value = 7 (simple test value)
        MVN   r0, r0                ; value = NOT value (invert bits)
        ADD   r0, r0, #1            ; value = value + 1 (add 1)

stop    B     stop
```

MVN instruction: **M**o**V**e and **N**egate
Moves a value from one register to another and negates (inverts) it

Note the syntax for ADD with immediate constant value #1

Recap: Negative Numbers

- **Sign-Magnitude**: Simply use the MSB to indicate sign, no longer used

  Issues: Complicated arithmetic and two representations of zero

- **Two's Complement**: Uses Modulo Arithmetic, most common method and what we will use
  - Range: $-(2^{N-1})$ to $+(2^{N-1} - 1)$
  - One representation of zero
  - Positive numbers are represented as in an unsigned number system
  - Negative numbers are represented by the 2's complement of their absolute value i.e. Invert and add one
  - *Advantage*: Arithmetic operations are identical to unsigned binary as long as the carry is discarded

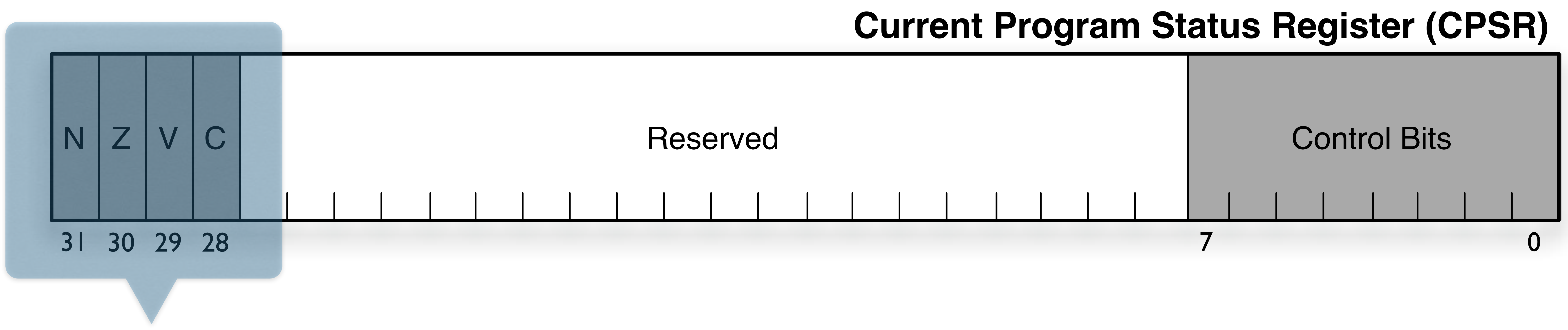
N.B.The computer does not know that we are storing negative numbers it is all up to our interpretation of the stored values

Condition Code Flags

Some instructions can optionally update the Condition Code Flags to provide information about the result of the execution of the instruction

These bits remember processing states between instructions.

e.g. whether the result of an addition was zero, or negative or whether a carry occurred



Condition Code Flags

N – Negative	Z – Zero
V – oVerflow	C – Carry

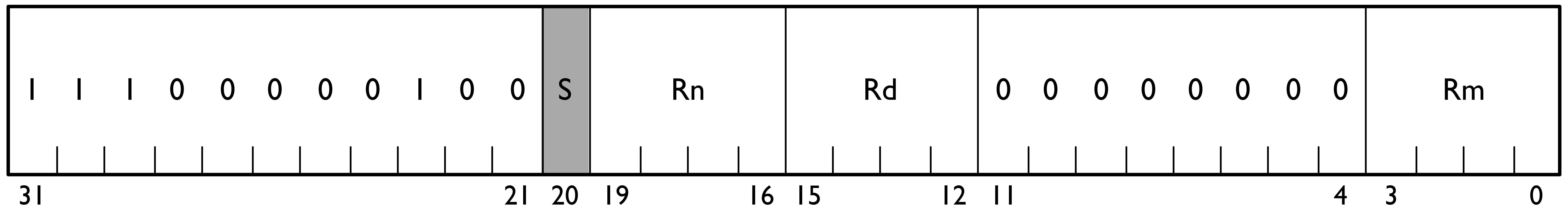
Condition Code Flags

The Condition Code Flags (N, Z, V, C) can be **optionally** updated to reflect the result of an instruction

S-bit in a machine code instruction is used to tell the processor whether the Condition Code Flags should be updated, based on the result

e.g. ADD instruction

Condition Code Flags only updated if S-bit (bit 20) is 1



In assembly language, we cause the Condition Code Flags to be updated by appending “S” to the instruction mnemonic (e.g. ADDS, SUBS, MOVS)

Negative (N) and Zero (Z) Flags

Zero condition Code Flag is (optionally) set if the result of the last instruction was exactly zero

Negative condition code flag is (optionally) set if the result of the last instruction was negative

i.e. If the Most Significant Bit (MSB) of the result is 1

Note that in Two's Complement negative numbers always have the first bit (MSB) set

If the MSB of the result is 1 the Negative flag will be set regardless of whether we are interpreting the values as signed or unsigned so it may not be relevant

Carry Flag (C)

```
start
        LDR   r0, =0xC0000000
        LDR   r1, =0x70000000
        ADDS  r0, r0, r1

stop    B     stop
```

ADDS causes the Condition Code Flags to be updated

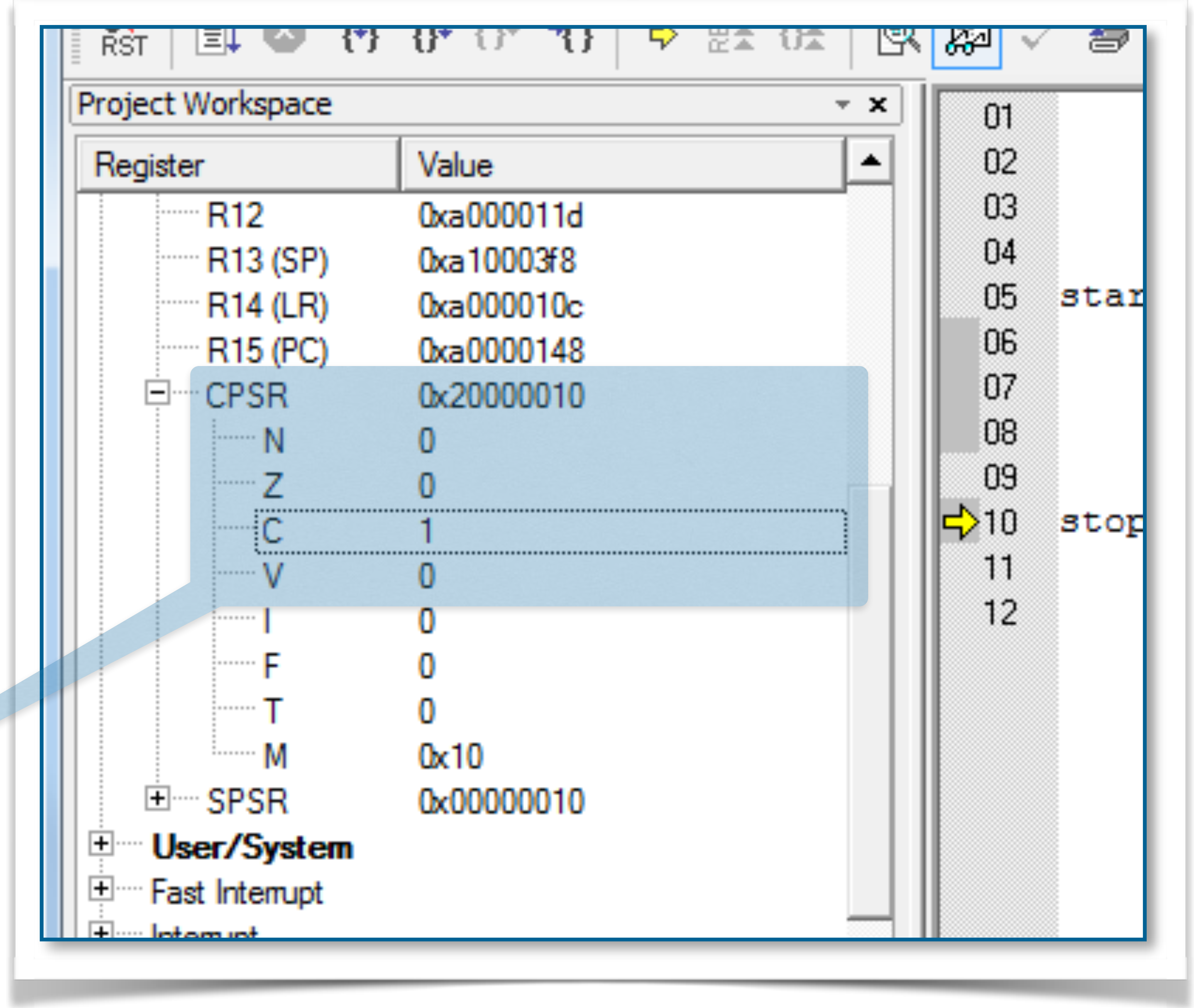
32-bit arithmetic

Expected result?

Does the result fit in 32-bits?

Will the carry flag be set?

Examine flags using µVision IDE

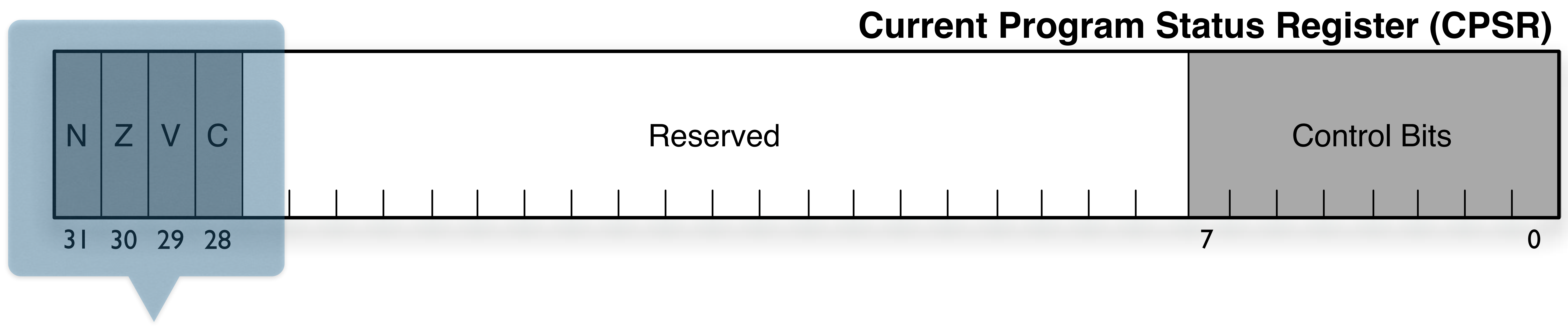


Register	Value
R12	0xa000011d
R13 (SP)	0xa10003f8
R14 (LR)	0xa000010c
R15 (PC)	0xa0000148
CPSR	0x20000010
N	0
Z	0
C	1
V	0
I	0
F	0
T	0
M	0x10
SPSR	0x00000010
User/System	
Fast Interrupt	

Overflow Flag (V)

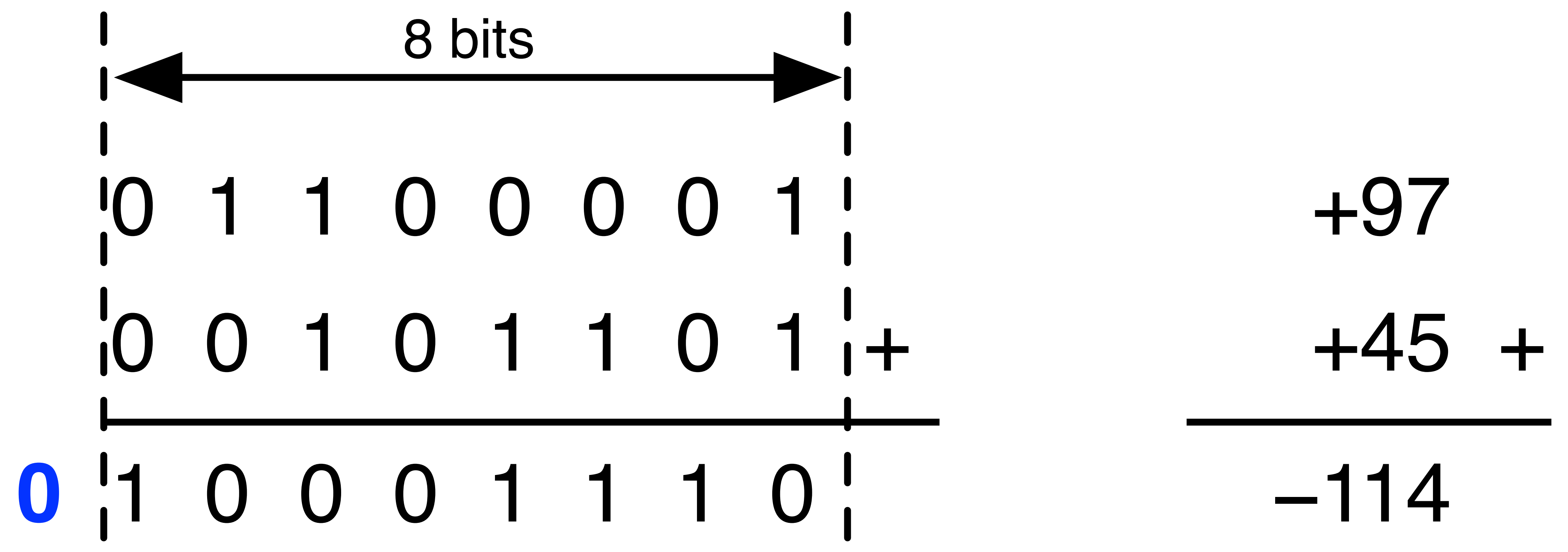
If the result of an addition or subtraction gives a result that is outside the range of the signed number system, then an oVerflow has occurred

The processor sets the oVerflow Condition Code Flag after performing an arithmetic operation to indicate whether an overflow has occurred



Condition Code Flags

Overflow Example



Result is 10001110_2 (142_{10}, or -114_{10})

If we were interpreting the two added values and the result as **signed integers**, we got an incorrect result:

We added two +ve numbers and obtained a –ve result

With 8-bits, the highest +ve integer we can represent is +127

10001110_2 (-114_{10})

The result is outside the range of the signed number system

Overflow Rules

Addition rule (r = a + b)

V = 1 if MSB(a) = MSB(b) and
MSB(r) ≠ MSB(a)

i.e. oVerflow accurs for addition if the operands have the same sign and the result has a different sign

Subtraction rule (r = a – b)

V = 1 if MSB(a) ≠ MSB(b) and
MSB(r) ≠ MSB(a)

i.e. oVerflow occurs for subtraction if the operands have different signs and the sign of the result is different from the sign of the first operand

Recap: Carry and Overflow

Carry and oVerflow flags always set by the processor regardless of our signed or unsigned interpretation of stored values

Processor does not “know” what our interpretation is

e.g. we could interpret the binary value 10001110_2 as either 142_{10} (unsigned) or -114_{10} (signed)

(we could also interpret it as the code for “Ä” or as the colour blue)

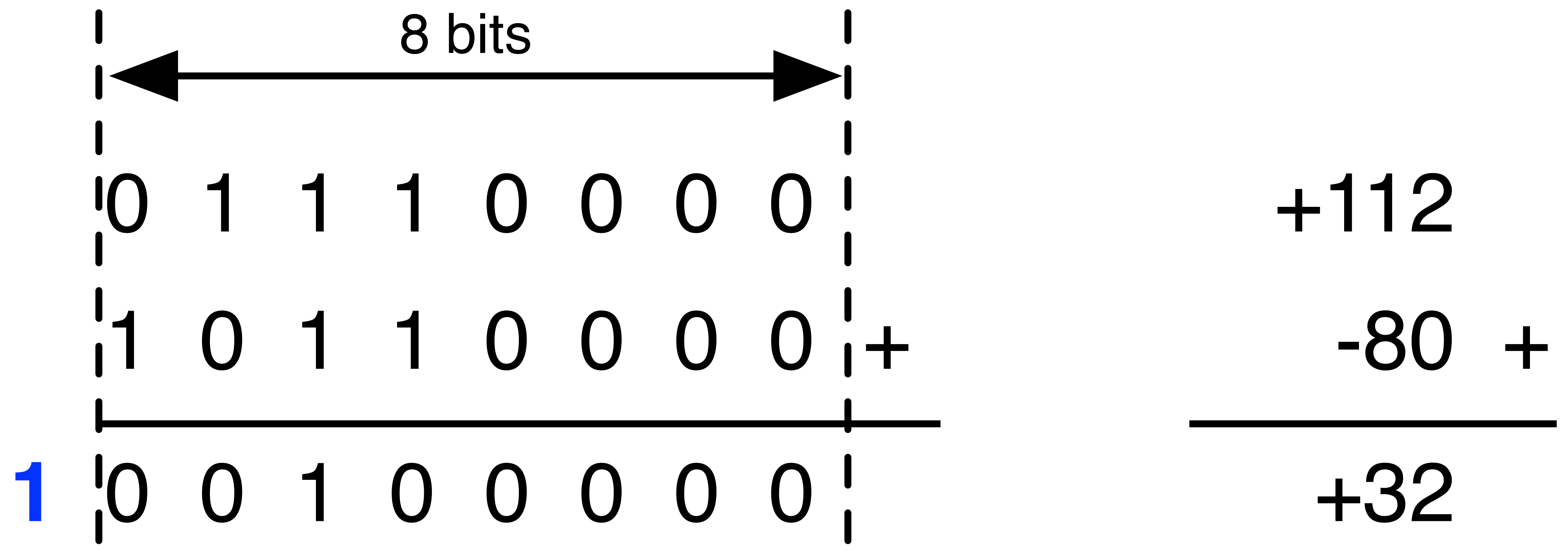
The C and V flags are set by the processor and it is our responsibility to choose

whether to interpret C or V (are we interpreting the values as unsigned or signed?)

how to interpret C or V

Generally: The carry flag is relevant to unsigned arithmetic whereas the overflow flag is relevant to signed arithmetic

Carry and Overflow Example



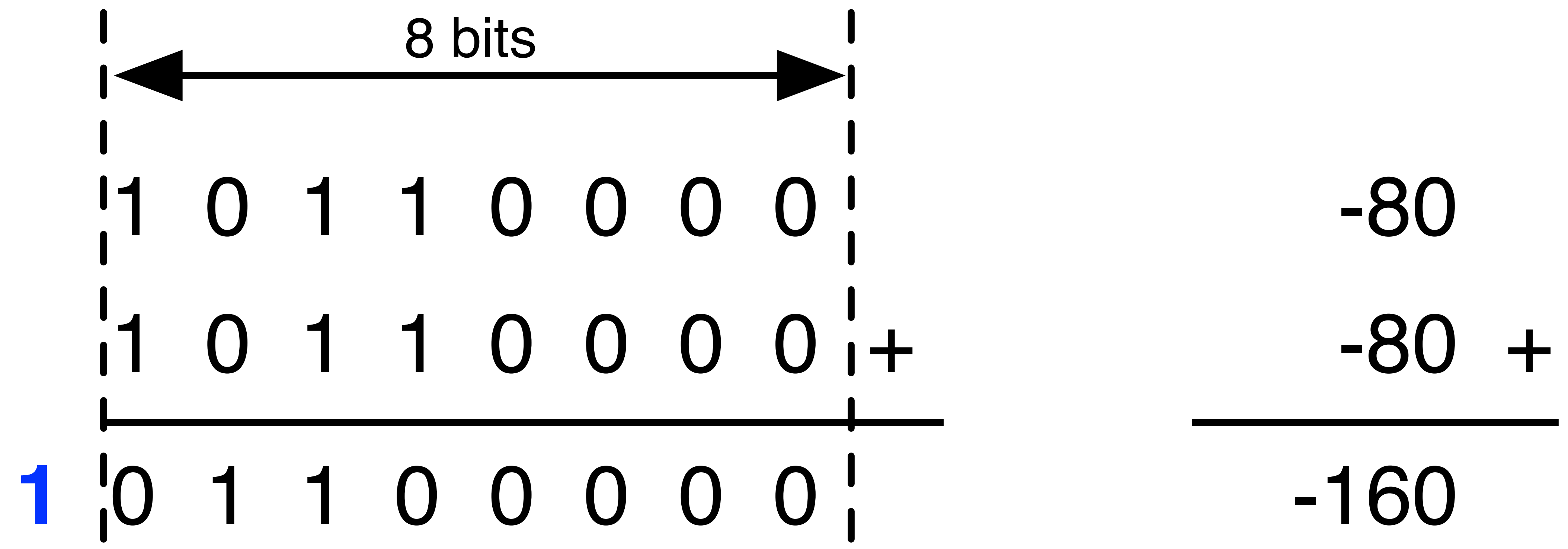
Signed interpretation: (+112) + (-80) = +32

Unsigned interpretation: 112 + 176 = 288

By examining the V flag (V = 0), we know that if were interpreting the values as signed integers, the result is correct

If we were interpreting the values as 8-bit unsigned values, C = 1 tells us that the result was too large to fit in 8-bits

Carry and Overflow Example



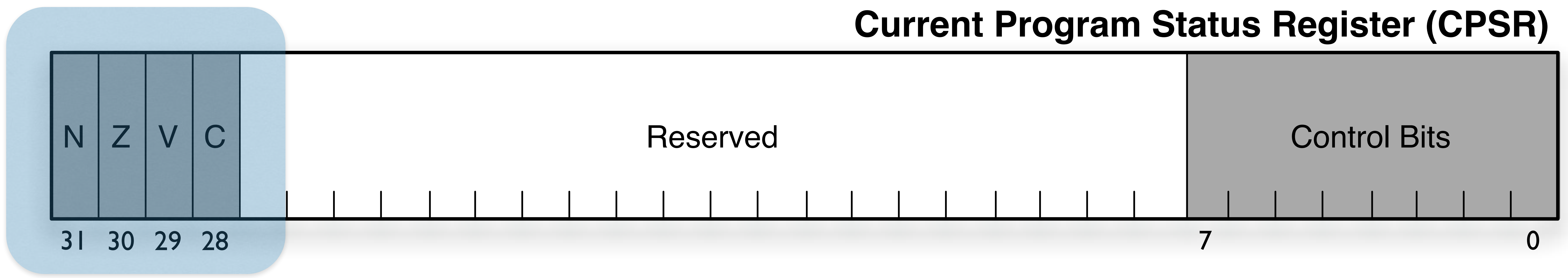
Signed: (-80) + (-80) = -160

Unsigned: 176 + 176 = 352

By examining the V flag (V = 1), we know that if were interpreting the values as signed integers, the result is outside the range of the signed number system

If we were interpreting the values as 8-bit unsigned values, C = 1 tells us that the result was too large to fit in 8-bits

Condition Code Flags – Recap



Many instructions can optionally cause the processor to update the Condition Code Flags (N, Z, V, and C) to reflect certain properties of the result of an operation

Append "S" to instruction in assembly language (e.g. ADDS)

Set S-bit in machine code instruction

N flag set to 1 if result is **N**egative (i.e. if MSB is 1)

Z flag is set to 1 if result is **Z**ero (i.e. all bits are 0)

C flag set if **C**arry occurs (addition) or borrow does not occur (subtraction)

V flag set if o**V**erflow occurs for addition or subtraction

Remember: Processor does this regardless of our interpretation of values as signed or unsigned