# Review

R102 – CompArch, Numbers, Dry Runs

## Starter – binary addition

- The method for binary addition is the same as in any other number base (including decimal), except we get *much* more carrying. For the addition of $3_{10}$ and $14_{10}$:

$$\begin{array}{r} 0\ 0\ 0\ 0\ \ 0\ 0\ 1\ 1 \\ 0\ 0\ 0\ 0\ \ 1\ 1\ 1\ 0 \\ \hline \\ \hline \end{array}$$

## Starter – binary addition

- The method for binary addition is the same as in any other number base (including decimal), except we get *much* more carrying. For the addition of $3_{10}$ and $14_{10}$:

$$\begin{array}{cccc|cccc} 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ \hline 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{array}$$

# Starter – binary addition

- The method for binary addition is the same as in any other number base (including decimal), except we get *much* more carrying. For the addition of $3_{10}$ and $14_{10}$:

$$\begin{array}{r} 0\,0\,0\,0\quad 0\,0\,1\,1 \\ 0\,0\,0\,0\quad 1\,1\,1\,0 \\ \hline 0\,0\,0\,1\quad 1\,1\,1\,1 \\ \hline \end{array}$$

- Try these:

$$\begin{array}{r} 0\,0\,0\,1\quad 1\,0\,1\,1 \\ 0\,0\,0\,0\quad 1\,1\,1\,1 \\ \hline \\ \hline \end{array}$$

$$\begin{array}{r} 0\,1\,1\,1\quad 0\,0\,1\,0 \\ 0\,1\,1\,1\quad 0\,0\,1\,0 \\ \hline \\ \hline \end{array}$$

$$\begin{array}{r} 1\,1\,0\,1\quad 1\,0\,0\,1 \\ 0\,0\,1\,1\quad 1\,1\,0\,1 \\ \hline \\ \hline \end{array}$$

# Starter – binary addition

- The method for binary addition is the same as in any other number base (including decimal), except we get *much* more carrying. For the addition of $3_{10}$ and $14_{10}$:

$$\begin{array}{r} 0\,0\,0\,0\quad 0\,0\,1\,1 \\ 0\,0\,0\,0\quad 1\,1\,1\,0 \\ \hline 0\,0\,0\,1\quad 1\,1\,1\,1 \end{array}$$

- Try these:

$$\begin{array}{r} 0\,0\,0\,1\quad 1\,0\,1\,1 \\ 0\,0\,0\,0\quad 1\,1\,1\,1 \\ \hline 0\,0\,1\,0\quad 1\,0\,1\,0 \end{array}$$

$$\begin{array}{r} 0\,1\,1\,1\quad 0\,0\,1\,0 \\ 0\,1\,1\,1\quad 0\,0\,1\,0 \\ \hline 1\,1\,1\,0\quad 0\,1\,0\,0 \end{array}$$

$$\begin{array}{r} 1\,1\,0\,1\quad 1\,0\,0\,1 \\ 0\,0\,1\,1\quad 1\,1\,0\,1 \\ \hline 0\,0\,0\,1\quad 0\,1\,1\,0 \end{array}$$

# Binary Addition and Subtraction & Two's Complement

Adding numbers together in base 2 and representing negative numbers

# Specification

### 4.5.4.2 Unsigned binary arithmetic

| Content |
| --- |
| Be able to add two unsigned binary integers |

### 4.5.4.3 Signed binary using two's complement

| Content | Additional information |
| --- | --- |
| Know that signed binary can be used to represent negative integers and that one possible coding scheme is two's complement. | This is the only representation of negative integers that will be examined. Students are expected to be able to convert between signed binary and decimal and vice versa. |
| Know how to:<br>• represent negative and positive integers in two's complement<br>• perform subtraction using two's complement<br>• calculate the range of a given number of bits, n. | |

# Overflow

- In a computer, binary numbers typically have a fixed length (number of bits)
- Arithmetic operations can, therefore, produce values that require more bits than we have space for
- This is called **overflow**
  - Often, we just ignore the overflow

# Discussion

How could we represent negative numbers in binary?

# Two’s complement

- This is the typical method for working with ‘signed’ numbers in binary
  - The sign of a number tells us whether it is positive or negative
  - A signed number can be either positive or negative, while an unsigned number is always positive

# Two’s complement

- This is the typical method for working with ‘signed’ numbers in binary
  - The sign of a number tells us whether it is positive or negative
  - A signed number can be either positive or negative, while an unsigned number is always positive
- Uses fixed-width binary numbers
- The most significant (leftmost) bit is the **sign bit**

# Two's complement

- This is the typical method for working with 'signed' numbers in binary
  - The sign of a number tells us whether it is positive or negative
  - A signed number can be either positive or negative, while an unsigned number is always positive
- Uses fixed-width binary numbers
- The most significant (leftmost) bit is the **sign bit**

| −128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
|---|---|---|---|---|---|---|---|
| | | | | | | | |

- If the sign bit is **1**, the number is **negative**
- If the sign bit is **0**, the number is **positive**

# Representing negative numbers in two’s complement

$$-45_{10}$$

| −128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
|---|---|---|---|---|---|---|---|

# Representing negative numbers in two’s complement

$$-45_{10}$$

| −128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
|---|---|---|---|---|---|---|---|
| 1 | | | | | | | |

# Representing negative numbers in two’s complement

$$-45_{10}$$

| −128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
|---|---|---|---|---|---|---|---|
| 1 | 1 | | | | | | |

# Representing negative numbers in two’s complement

$$-45_{10}$$

| −128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 0 | | | | | |

# Representing negative numbers in two’s complement

$$-45_{10}$$

| −128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 0 | 1 | | | | |

# Representing negative numbers in two’s complement

$$-45_{10}$$

| −128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 0 | 1 | 0 | | | |

# Representing negative numbers in two’s complement

$$-45_{10}$$

| −128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 0 | 1 | 0 | 0 | | |

# Representing negative numbers in two’s complement

$$-45_{10}$$

| −128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 0 | 1 | 0 | 0 | 1 | |

# Representing negative numbers in two’s complement

$$-45_{10}$$

| −128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 0 | 1 | 0 | 0 | 1 | 1 |

# Representing negative numbers in two’s complement

$$-45_{10}$$

| −128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 0 | 1 | 0 | 0 | 1 | 1 |

Converting back to decimal is just the same as unsigned binary, but remember that the first column is negative:

$$-128 + 64 + 16 + 2 + 1 = -45$$

## Practice

Convert −14 into two’s complement binary using 8 bits

## Practice

Convert –14 into two's complement binary using 8 bits

1111 0010

## Practice

Convert –127 into two’s complement binary using 8 bits

## Practice

Convert –127 into two's complement binary using 8 bits

1000 0001

# Practice

Convert 1011 1000 from two’s complement binary to decimal

# Practice

Convert 1011 1000 from two’s complement binary to decimal

−72

# Practice

Convert 0011 1000 from two’s complement binary to decimal

# Practice

Convert 0011 1000 from two's complement binary to decimal

56

# 'Flipping the sign'

- Consider the number −44 in binary – how would we turn it into 44?

| −128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 0 | 1 | 0 | 1 | 0 | 0 |

- **Trick:** Starting at the right, find all bits up to and including the first 1. Keep these bits the same, and flip the remaining bits.
- This works in reverse too

# 'Flipping the sign'

- Consider the number −44 in binary – how would we turn it into 44?

| −128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 0 | 1 | 0 | 1 | 0 | 0 |

- **Trick:** Starting at the right, find all bits up to and including the first 1. Keep these bits the same, and flip the remaining bits.
- This works in reverse too

# 'Flipping the sign'

- Consider the number −44 in binary – how would we turn it into 44?

| −128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 |

- **Trick:** Starting at the right, find all bits up to and including the first 1. Keep these bits the same, and flip the remaining bits.
- This works in reverse too

# Binary subtraction

- Suppose we want to perform 50 – 29 in binary
  - From GCSE maths, we know that 50 – 29 = 50 + (–29)
- Subtraction is just the same as adding negative numbers
  - This is particularly useful in binary

# Binary subtraction

- Suppose we want to perform 50 – 29 in binary
  - From GCSE maths, we know that 50 – 29 = 50 + (–29)
- Subtraction is just the same as adding negative numbers
  - This is particularly useful in binary

$50_{10}$ = 0011 0010

$-29_{10}$ = 1110 0011

- We can use binary addition to add these together, ignoring the overflow

# Binary subtraction

- Suppose we want to perform 50 – 29 in binary
  - From GCSE maths, we know that 50 – 29 = 50 + (–29)
- Subtraction is just the same as adding negative numbers
  - This is particularly useful in binary

$50_{10}$ = 0011 0010

$-29_{10}$ = 1110 0011

- We can use binary addition to add these together, ignoring the overflow
  - 0001 0101 = $21_{10}$

# Example 1

$$9 - 29$$

$9_{10} = 0000\ 1001$

$-29_{10} = 1110\ 0011$

# Example 1

$$9 - 29$$

$9_{10} = 0000\ 1001$

$-29_{10} = 1110\ 0011$

$1110\ 1100 = -20_{10}$

# Example 2

$$100 + 50$$

$100_{10} = 0110\ 0100$

$50_{10} = 0011\ 0010$

# Example 2

$$100 + 50$$

$100_{10} = 0110\ 0100$

$50_{10} = 0011\ 0010$

1001 0110 = –106 (not 150?)

# Example 2

$$100 + 50$$

$100_{10}$ = 0110 0100

$50_{10}$ = 0011 0010

1001 0110 = –106 (not 150?)

- With one byte using two's complement, we can only represent the range

–128 to 127

# Notes & practice

- Binary addition is the same as in decimal but with more **carrying**
- Computers often use fixed-width arithmetic where overflow is ignored
- Two's complement example: –29
  - Write the column headers with a negative sign bit header on the left
  - Keep adding numbers till we're done
  - Or, start positive and use the 'flipping the sign' method
- Subtraction in binary is just addition with two's complement negative numbers
- With 8 bits, we can only represent integers in the range –128 to 127 in two's complement integers

Try these problems:

1. Convert –56 to binary
2. Convert 1001 1111 into decimal
3. Compute 7 – 100 in binary, showing your working
4. Using two's complements and 2 bits after the binary point, compute 10 – 19.25 in binary, showing your working
5. What are the largest and smallest integers that can be represented using two's complement with two bytes?

# Answers

1. Convert −56 to binary
   - 1001 1000
2. Convert 1001 1111 into decimal
   - −97
3. Compute 7 − 100 in binary, showing your working
   - 1010 0011
4. Using two's complements and 2 bits after the binary point, compute 10 − 19.25 in binary, showing your working
   - 110110.11
5. What are the largest and smallest integers that can be represented using two's complement with two bytes?
   - $-32,768$ to $32,767$

# Random Numbers in C#

# Discussion

In pairs, discuss answers to these questions and write your ideas on a whiteboard.

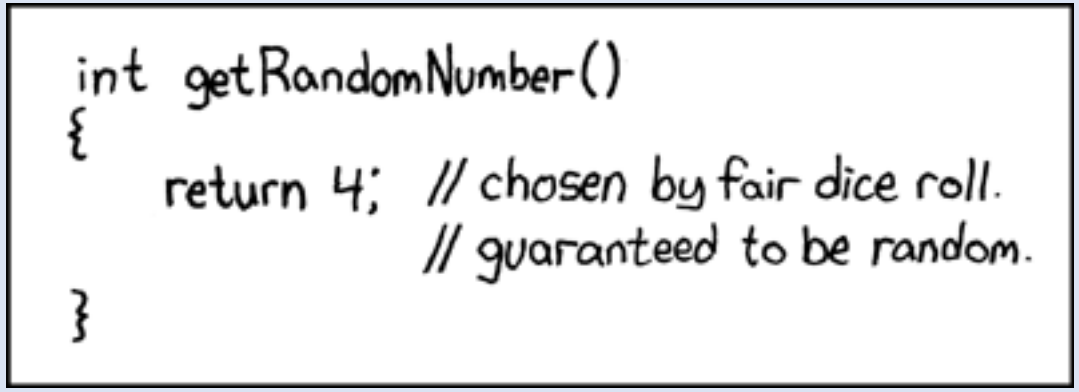
1. What is randomness?
2. What is a **random** number?
3. Can computers generate random numbers?
4. Why might we want to generate random numbers?

# “True” randomness?

- Tossing a coin
- Rolling a die
- Roulette wheel
- Decay of an atom

# Pseudorandom number generators

- A **pseudorandom number generator** (PRNG) is an algorithm that generates an apparently/observably random **sequence** of numbers



# Seeding PRNGs

- A **seed** is a number used to initialise a pseudorandom number generator
- Given the same seed, two PRNGs will produce the same sequence of ‘random’ numbers

# Seeding PRNGs

- A **seed** is a number used to initialise a pseudorandom number generator
- Given the same seed, two PRNGs will produce the same sequence of ‘random’ numbers
- Operating systems look for **entropy** (randomness through unpredictability)
  - Mouse movements
  - Keyboard timings
  - Variance in fan noise
- Cloudflare use an image feed of a rack of ~100 lava lamps to generate entropy

# Random numbers in C#

Generating a random integer:

```
Random random = new Random(); // Seeded automatically
Console.WriteLine(random.Next(1, 11));
```

This generates a random number from 1 to 10.

# Ranges of random numbers

What ranges will these produce?

- `random.Next(1, 21);`
- `random.Next(1, 201);`
- `random.Next(2000, 2099);`
- `2 * random.Next(1, 51);`
- `Math.Pow(2, random.Next(1, 9));`

# Ranges of random numbers

What ranges will these produce?

- `random.Next(1, 21);`
- `random.Next(1, 201);`
- `random.Next(2000, 2099);`
- `2 * random.Next(1, 51);`
- `Math.Pow(2, random.Next(1, 9));`

- 1 to 20

# Ranges of random numbers

What ranges will these produce?

- `random.Next(1, 21);`
- `random.Next(1, 201);`
- `random.Next(2000, 2099);`
- `2 * random.Next(1, 51);`
- `Math.Pow(2, random.Next(1, 9));`

- 1 to 20
- 1 to 200

# Ranges of random numbers

What ranges will these produce?

- `random.Next(1, 21);`
- `random.Next(1, 201);`
- `random.Next(2000, 2099);`
- `2 * random.Next(1, 51);`
- `Math.Pow(2, random.Next(1, 9));`

- 1 to 20
- 1 to 200
- 2000 to 2098

# Ranges of random numbers

What ranges will these produce?

- `random.Next(1, 21);`
- `random.Next(1, 201);`
- `random.Next(2000, 2099);`
- `2 * random.Next(1, 51);`
- `Math.Pow(2, random.Next(1, 9));`

- 1 to 20
- 1 to 200
- 2000 to 2098
- Even numbers, 2 to 100

# Ranges of random numbers

What ranges will these produce?

- `random.Next(1, 21);`
- `random.Next(1, 201);`
- `random.Next(2000, 2099);`
- `2 * random.Next(1, 51);`
- `Math.Pow(2, random.Next(1, 9));`

- 1 to 20
- 1 to 200
- 2000 to 2098
- Even numbers, 2 to 100
- Powers of 2, 2 to 256

# Reusing random number generators

- It is good practice to reuse a single random number generator
  - The version of C# we use uses the system clock to seed PRNGs, so creating multiple in close succession can lead to duplicate “random” numbers
  - It is always good practice to reuse them

```
Random random = new Random();
int num1 = random.Next(1, 11);
int num2 = random.Next(1, 11);
```

# Practice

Example:

```
Random random = new Random();
for (int i = 0; i < 20; i++)
{
  Console.WriteLine(random.Next(1, 11));
}
```

You have **5 minutes** to see how far you can get.
Produce a list of 20 random numbers between:

1. 1 to 30
2. 51 to 60
3. Even numbers, 0 to 10
4. Odd numbers, 1 to 11
5. Sums of powers of 3 and 5 up to $3^4 + 5^4$

# Chance

- Sometimes, we are interested in probability
- We may have a random event that we want to occur 40% of the time in a game (e.g. a critical hit)

```
Random random = new Random();
int value = random.NextDouble();
```

- `random.NextDouble()` generates a decimal $x$ such that $0 \leq x < 1$

# Worksheet

## W105 - C# Random Numbers