Lab 04

Part A – Modulo

Description

This activity simulates an implementation of the modulo operator.

Procedure

Background

- The components of an integer division operation are defined as follows, where $\mathbb{Z}$ is the set of integers:
  - Dividend $a \in \mathbb{Z}$
  - Divisor $d \in \mathbb{Z}^+$
  - Quotient $q \in \mathbb{Z}$
  - Remainder $r \in \mathbb{Z}$, $0 \leq r < d$
- The *modulo* function or operator is defined as:
  - $r = a \bmod d$
    - in this case d is the *modulus*, which is generally represented as m or n
- Modulo is distributive over addition, subtraction, multiplication, and division.

$$(a + b) \bmod m = \big[(a \bmod m) + (b \bmod m)\big] \bmod m$$

$$(a - b) \bmod m = \big[(a \bmod m) - (b \bmod m)\big] \bmod m$$

$$ab \bmod m = \big[(a \bmod m)(b \bmod m)\big] \bmod m$$

$$\frac{a}{b} \bmod m = \big[(a \bmod m)(b^{-1} \bmod m)\big] \bmod m \quad \text{iff. } \gcd(b, n) = 1$$

  - However, division is only defined when b and m are coprime, as this is the only time b^{-1} (the inverse of b under modulo m) exists.
  - Modulo is also distributive over *exponentiation*. This leads to a highly useful application of modulo called fast exponentiation.

$$a^k \bmod m = (a \bmod m)^k \bmod m$$

- The modulo function always results in a positive number in a *finite field* $\mathbb{Z}_m$, which is the set of non-negative integers less than m, $\{0, 1, \ldots, m-1\}$.
  - If the remainder of integer division would be negative, you must keep adding m until you acquire a positive number, which will be the result of the operation.
  - This is useful for *encryption* to prevent an encrypted message from having a different range than the message, confusing would-be attackers.
- Within the finite field $\mathbb{Z}_m$, a form of arithmetic *modulo* m is defined.
  - Operations such as addition, subtraction, multiplication, and division are possible inside this finite field.

Algorithm for Computing $a \bmod m$

1. If $a < m$, the algorithm ends and the final output is a.
2. If $a \geq n$, then the algorithm initializes the output first to a.
3. The algorithm defines a variable y as the location of the most significant high bit of m.
4. The algorithm defines a variable x as the location of the most significant high bit of the output.
5. The algorithm shifts m to the left by adding $(x - y)$ zeros to m after its least significant bit.
6. If the shifted version of m is larger than the output, the algorithm will repeat the previous step (5) using $x - y - 1$ zeros instead.
7. The algorithm calculates the new output by subtracting the shifted version of m from the previous output.
8. If the output is larger than m, then the algorithm repeats steps 4-7. If not, then the algorithm ends and $(a \bmod m)$ is the final output.

- An implementation of the algorithm can be found here: https://pl.kotl.in/fbgy24OBH

Activity

1. Calculate $30 \bmod 3$, $50 \bmod 7$, and $85 \bmod 11$ using regular division (by hand).
2. Simulate the algorithm using the provided program and calculate the examples in Step 1 to verify your answers.
3. The program will also provide the results of the built-in function `mod()`.

Discussion

4. Compare the results of steps 1, 2, and 3. Are they different? If so, how?
5. What components do we need to implement this algorithm in hardware?
6. If a is a negative value, would the algorithm still work? If not, what do we need to change?

Deliverables

- Include answers to the discussion questions (4, 5, and 6) from this activity in your **informal report** for Lab 04.

Outcomes

- Understand how the modulo operation is implemented in hardware.