TP2

Daniel Francisco Texeira Andrade - A100057

Pedro André Ferreira Malainho - A100050

Problema 1

Enunciado

Considere a descrição da cifra A5/1 que consta no documento +Lógica Computacional: a Cifra A5/1. Informação complementar pode ser obtida no artigo da Wikipedia.

Pretende-se

a. Definir e codificar, em Z3 e usando o tipo BitVector para modelar a informação, uma FSM que descreva o grador de chaves.

b. Considere as seguintes eventuais propriedades de erro:

i. ocorrência de um "*burst*" 0^t (*t* zeros) que occorrem em 2^t passos ou menos

ii. ocorrência de um "*burst*" de tamanho *t* que repete um "*burst*" anterior no mesmo output em $2^{t/2}$ passos ou menos.

Tente codificar estas propriedades e verificar se são acessíveis a partir de um estado inicial aleatoriamente gerado.

Referências

- +Wikipedia
- +Implementação pedagógica de uma cifra A5/1 (Linguagem: C)

Importes

In [1]:
```
from z3 import *
from random import randint
from pysmt.shortcuts import *
```

Implementação

Declare_State Function

This function declares symbolic representations of three Linear Feedback Shift Registers (LFSRs) for a given time step (i). Each LFSR has a distinct bit width, and the function returns a dictionary with symbolic variables representing the state of each LFSR at that time step.

`Parameters:`

- `i` : An integer representing the current time step.

`Returns:` A dictionary containing:

- `'lfsr0'` : A `BitVec` representing the first LFSR state at step (i), with bit width `R0MASK` .
- `'lfsr1'` : A `BitVec` representing the second LFSR state at step (i), with bit width `R1MASK` .
- `'lfsr2'` : A `BitVec` representing the third LFSR state at step (i), with bit width `R2MASK` .

Get_Output Function

This function calculates the output bit for a given LFSR state dictionary by extracting the most significant bit (MSB) from each LFSR. It then computes the XOR of these bits to produce a single output bit.

`Parameters:`

- `state` : A dictionary containing the current state of the LFSRs, where keys `'lfsr0'` , `'lfsr1'` , and `'lfsr2'` correspond to each LFSR's current state as a `BitVec` .

`Returns:` A logical expression representing the XOR of the MSB from each LFSR, giving the overall output bit for the given state.

Transition Function

This function defines the transition conditions between two consecutive LFSR states, `curr` (current state) and `nxt` (next state). It calculates the control bits for each LFSR, determines the majority bit from these control bits, and applies conditions that each LFSR should transition to its next state if its control bit matches the majority bit. Each LFSR transition depends on a bitwise shift and an XOR operation with the respective `S0` , `S1` , or `S2` initialization values.

`Parameters:`

- `curr` : A dictionary representing the current state of the LFSRs.
- `nxt` : A dictionary representing the next state of the LFSRs.

`Returns:` A logical `Or` expression representing the allowed transitions based on the majority bit condition. At least two LFSRs must transition at each step, and the function enforces valid transitions for each register by checking that the control bit aligns with the majority bit.

Check_Bursts Function

This function verifies if a certain type of burst occurs in the LFSR output sequence. A burst is defined as a specific pattern in the output, such as a sequence of zeros (`"first"` burst) or a repeated sequence of bits (`"second"` burst). The function constructs constraints based on the desired burst type, adds them to the solver, and checks if they are satisfiable.

`Parameters:`

- `t` : Integer specifying the length of the burst pattern to check.
- `states` : A list of dictionaries, each representing the symbolic state of the LFSRs at a given step.
- `outputs` : A list of logical expressions representing the output bit at each time step.
- `solver` : A `Solver` instance used to evaluate the constraints.
- `burst_type` : (Optional) A string specifying the burst type to check. `"first"` checks for a burst of `t` consecutive zeros, while `"second"` checks for a repeated pattern of length `t` .

`Returns:` A boolean indicating whether the burst type specified by `burst_type` is reachable within the sequence. If a burst is reachable, the function prints an example LFSR state leading to the burst pattern.

In [48]:
```
from z3 import *

R0MASK, R1MASK, R2MASK = 19, 22, 23

cbit0, cbit1, cbit2 = 8, 10, 10

S0 = BitVec('S0', R0MASK)
S1 = BitVec('S1', R1MASK)
S2 = BitVec('S2', R2MASK)


def declare_state(i):

    return {
        'lfsr0': BitVec(f'lfsr0_{i}', R0MASK),
        'lfsr1': BitVec(f'lfsr1_{i}', R1MASK),
        'lfsr2': BitVec(f'lfsr2_{i}', R2MASK)
    }
```

```
def get_output(state):

    return Extract(R0MASK - 1, R0MASK - 1, state['lfsr0']) ^ \
           Extract(R1MASK - 1, R1MASK - 1, state['lfsr1']) ^ \
           Extract(R2MASK - 1, R2MASK - 1, state['lfsr2'])


def transition(curr, nxt):

    c0 = Extract(cbit0, cbit0, curr['lfsr0'])
    c1 = Extract(cbit1, cbit1, curr['lfsr1'])
    c2 = Extract(cbit2, cbit2, curr['lfsr2'])
    majority_bit = (c0 & c1) | (c1 & c2) | (c0 & c2)

    # Transition conditions for each LFSR based on the majority bit
    t0 = And(c0 == majority_bit, nxt['lfsr0'] == (curr['lfsr0'] << 1) ^ (curr['l
    t1 = And(c1 == majority_bit, nxt['lfsr1'] == (curr['lfsr1'] << 1) ^ (curr['l
    t2 = And(c2 == majority_bit, nxt['lfsr2'] == (curr['lfsr2'] << 1) ^ (curr['l

    return Or(And(t0, t1), And(t0, t2), And(t1, t2), And(t0, t1, t2))


def check_bursts(t, states, outputs, solver, burst_type="first"):

    max_steps = (1 << t) if burst_type == "zero" else (1 << (t // 2))
    conditions = []

    if burst_type == "first":
        for i in range(max_steps - t + 1):
            conditions.append(And([outputs[i + j] == 0 for j in range(t)]))
    elif burst_type == "second":
        for i in range(max_steps - t + 1):
            for j in range(i + t, max_steps - t + 1):
                conditions.append(And([outputs[i + k] == outputs[j + k] for k in
    else:
        pass

    solver.add(Or(conditions))
    result = solver.check() == sat

    if result:
        model = solver.model()
        print(f"\nExample state leading to a {burst_type} burst:")
        for lfsr, mask in zip(['lfsr0', 'lfsr1', 'lfsr2'], [R0MASK, R1MASK, R2MA
            print(f"\t{lfsr}: {format(model[states[0][lfsr]].as_long(), f'0{mask
    return result


def bmc_always(t):

    max_steps = max(1 << t, 1 << (t // 2))

    states = [declare_state(i) for i in range(max_steps)]
    outputs = [get_output(state) for state in states]

    solver = Solver()

    for i in range(max_steps - 1):
```

```
        solver.add(transition(states[i], states[i + 1]))

    # Check for zero burst
    solver.push()
    zero_burst = check_bursts(t, states, outputs, solver, "first")
    solver.pop()

    # Check for repeated burst
    solver.push()
    repeat_burst = check_bursts(t, states, outputs, solver, "second")
    solver.pop()

    print(f"\nBurst analysis for initial arbitrary state:")
    print(f"Burst of {t} consecutive zeros reachable within 2^{t} steps: {zero_b
    print(f"Repeated burst of length {t} reachable within 2^{t // 2} steps: {rep


bmc_always(4)
```

```
Example state leading to a first burst:
        lfsr0: 0111011111111111111
        lfsr1: 11100101110101011110000
        lfsr2: 11011100011110011101111

Burst analysis for initial arbitrary state:
Burst of 4 consecutive zeros reachable within 2^4 steps: True
Repeated burst of length 4 reachable within 2^2 steps: False
```

2ª Implementação ????

In [254…

```
# Masks for the three shift registers
R1MASK = 0x07FFFF   # 19 bits, numbered 0..18
R2MASK = 0x3FFFFF   # 22 bits, numbered 0..21
R3MASK = 0x7FFFFF   # 23 bits, numbered 0..22

# Feedback taps, for clocking the shift registers.
# These correspond to the primitive polynomials
# x^19 + x^5 + x^2 + x + 1,
# x^22 + x + 1,
# and x^23 + x^15 + x^2 + x + 1.
R1TAPS = 0x072000 # bits 18, 17, 16, 13
R2TAPS = 0x300000 # bits 21, 20
R3TAPS = 0x700080 # bits 22, 21, 20, 7
```

Clock_One Function

This function performs a single clock cycle on a shift register using linear feedback shift register (LFSR) logic. The register is advanced by one step, with feedback applied according to the polynomial defined by TAPS.

`Parameters:`

- `R:` The current state of the shift register as an integer.

- `MASK:` A bitmask specifying the number of bits in the register. This ensures that only the allowed bits in the register are kept.
- `TAPS:` Feedback taps for the LFSR, specified as a bitmask. Each bit set in TAPS indicates a feedback tap point in the polynomial.

`Returns:` The updated state of the register after one clock cycle.

In [255…
```
# Steps a shift register forward by one, based on feedback taps
def clock_one(R, MASK, TAPS):
    new_bit = 0
    for i in range(23):  # Adjust to the largest number of bits (R3 has 23)
        if (R & (1 << i)) != 0:
            new_bit ^= (TAPS & (1 << i)) != 0
    new_bit &= 1
    R = ((R << 1) & MASK) | new_bit
    return R
```

Generate_Key function

This function extracts certain bits from the current states of the three shift registers, R1, R2, and R3, to generate a key sequence.

`Parameters:`

- `R1:` The current state of the first register.
- `R2:` The current state of the second register.
- `R3:` The current state of the third register.

`Returns:` A list of specific bits taken from the three registers, forming a key sequence.

In [256…
```
# Generate a key from the current states of the registers
def generate_key(R1, R2, R3):
    return [
        R1 & 1,              # LSB of R1
        (R1 >> 1) & 1,       # Bit 1 of R1
        (R1 >> 2) & 1,       # Bit 2 of R1
        (R2 >> 9) & 1,       # Bit 9 of R2
        (R2 >> 10) & 1,      # Bit 10 of R2
        (R3 >> 20) & 1,      # Bit 20 of R3
        (R3 >> 21) & 1,      # Bit 21 of R3
        (R3 >> 22) & 1       # MSB of R3
    ]
```

Check_For_Zero_Burst fucntion

This function simulates the shift registers over a given number of steps and checks for a burst of consecutive zeros of length t within a sliding window of the last 2^t output bits.

`Parameters:`

- `steps:` The number of steps to simulate.

- `t:` The target burst length (number of consecutive zeros) to detect.
- `R1, R2, R3:` The initial states of the three shift registers.

`Returns:`

- A tuple (zero_burst, step) if a burst of zeros of length t is found, where zero_burst is the zero sequence found, and step is the step at which it was found.
- None if no zero burst of the specified length is found within the given steps.

In [257…

```
# Check for bursts of zeros of size t that occur within 2^t steps
def check_for_zero_burst(steps, t, R1, R2, R3):
    output_bits = []
    step_limit_zero = 2 ** t
    zero_burst = '0' * t  # Burst of t zeros

    for step in range(steps):
        key = generate_key(R1, R2, R3)
        output_bits.extend(key)

        # Clock the registers
        R1, R2, R3 = clock_one(R1, R1MASK, R1TAPS), clock_one(R2, R2MASK, R2TAPS

        # Check for burst of zeros of size t within the most recent 2^t bits
        if ''.join(map(str, output_bits[-t:])) == zero_burst:
            return zero_burst, step

        # Limit output_bits length to the most recent step_limit_zero bits to sa
        if len(output_bits) > step_limit_zero + t:
            output_bits = output_bits[-(step_limit_zero + t):]

    return None
```

Check_For_Burst_Repetition function

This function checks for repeating bursts of bits of length t within a window of $2^{\frac{t}{2}}$ steps. It looks for burst patterns that reappear within a shorter range than in check_for_zero_burst.

`Parameters:`

- `steps:` The number of steps to simulate.
- `t:` The target burst length to detect for repetitions.
- `R1, R2, R3:` The initial states of the three shift registers.

`Returns:`

- A tuple (burst, first_step, second_step) if a repeating burst is found, where burst is the repeated sequence, first_step is the first occurrence, and second_step is the second occurrence within the required range.
- None if no repeating burst of the specified length is found within the given steps.

In [258…

```
# Check for bursts of size t that repeat within 2^(t/2) steps
def check_for_burst_repetition(steps, t, R1, R2, R3):
```

```
    output_bits = []
    burst_dict = {}
    step_limit_repetition = 2 ** (t // 2)

    for step in range(steps):
        key = generate_key(R1, R2, R3)
        output_bits.extend(key)

        # Clock the registers
        R1, R2, R3 = clock_one(R1, R1MASK, R1TAPS), clock_one(R2, R2MASK, R2TAPS

        # Get the last t bits as a burst
        burst = ''.join(map(str, output_bits[-t:]))

        # Check if burst is in the dictionary and within the required steps
        if burst in burst_dict:
            previous_step = burst_dict[burst]
            if step - previous_step <= step_limit_repetition:
                return burst, previous_step, step
        burst_dict[burst] = step

        # Limit output_bits length to recent 2^(t/2) bursts to save memory
        if len(output_bits) > step_limit_repetition + t:
            output_bits = output_bits[-(step_limit_repetition + t):]

    return None
```

Exemplos

In [7]:

```
# Function to verify burst properties using Z3
def solve_burst_properties(t):
    # Create the state variables
    R1 = BitVec('R1', 19)
    R2 = BitVec('R2', 22)
    R3 = BitVec('R3', 23)

    # Initialize registers with random non-zero values
    initial_R1 = randint(1, R1MASK)
    initial_R2 = randint(1, R2MASK)
    initial_R3 = randint(1, R3MASK)

    # Define the solver and add constraints for initial non-zero state
    solver = Solver()
    solver.add(R1 == initial_R1, R2 == initial_R2, R3 == initial_R3)
    solver.add(And(R1 != 0, R2 != 0, R3 != 0))

    # Check if properties are reachable
    if solver.check() == sat:
        model = solver.model()
        print('\x1b[6;30;42m' + "Properties are reachable from the initial state
        print(f"R1: {model[R1]}, R2: {model[R2]}, R3: {model[R3]}")

        # Generate and print the key
        key = generate_key(model[R1].as_long(), model[R2].as_long(), model[R3].a
        print("Generated Key:", ''.join(map(str, key)))

        # Check for bursts of zeros
```

```
        # steps = 1000
        steps = 2^t
        result = check_for_zero_burst(steps, t, model[R1].as_long(), model[R2].a

        if result:
            zero_burst, found_step = result
            print(f"A burst of zeros of size {t} was found within {2**t} steps:
            print(f"It was found at step {found_step}.")
        else:
            print(f"No repeating burst of zeros of size {t} was found within {2*

        steps = 2^(t//2)

        # Check for bursts of repetition
        found_burst_info = check_for_burst_repetition(steps, t, model[R1].as_lon

        if found_burst_info:
            burst, first_step, second_step = found_burst_info
            print(f"A burst of size {t} was found that repeats within {2**(t//2)
            print(f"It was found at steps {first_step} and {second_step}.")
        else:
            print(f"No repeating burst of size {t} was found within {2**(t//2)}
    else:
        print("Properties are NOT reachable")
```

Exemplo 1

Caracteristicas:

```
R1, R2, R3 = randint(1, RMASK)
t = 3
```

In [12]:
```
solve_burst_properties(t=3)
```

```
Properties are reachable from the initial state:
R1: 247989, R2: 1181177, R3: 179176
Generated Key: 10101000
A burst of zeros of size 3 was found within 8 steps: 000.
It was found at step 0.
A burst of size 3 was found that repeats within 2 steps: 000.
It was found at steps 0 and 1.
```

Exemplo 2

Caracteristicas:

```
R1, R2, R3 = randint(1, RMASK)
t = 15
```

In [153…
```
solve_burst_properties(t=15)
```

```
Properties are reachable from the initial state:
R1: 481131, R2: 2989596, R3: 4742438
Generated Key: 11011001
No repeating burst of zeros of size 15 was found within 32768 steps.
No repeating burst of size 15 was found within 128 steps.
```

Exemplo 3

Caracteristicas:

R1 == 0b10101010101010101010, equivalente a 349525
R2 == 0b11001100110011001100110011, equivalente a 3355443
R3 == 0b11111111111111111111111, equivalente a 8388607
t = 2

In [154…

```python
# Function to verify burst properties using Z3
def solve_burst_properties(t):
    # Create the state variables
    R1 = BitVec('R1', 19)
    R2 = BitVec('R2', 22)
    R3 = BitVec('R3', 23)

    # Initialize registers with specific pattern values
    solver = Solver()
    solver.add(R1 == 0b10101010101010101010, R2 == 0b11001100110011001100110011, R3 =

    # Check properties: verify output patterns
    solver.add(And(R1 != 0, R2 != 0, R3 != 0))

    # Check if properties are reachable
    if solver.check() == sat:
        model = solver.model()
        print('\x1b[6;30;42m' + "Properties are reachable from the initial state
        print(f"R1: {model[R1]}, R2: {model[R2]}, R3: {model[R3]}")

        # Generate and print the key
        key = generate_key(model[R1].as_long(), model[R2].as_long(), model[R3].a
        print("Generated Key:", ''.join(map(str, key)))

        # Check for bursts of zeros
        steps = 2^(t)
        result = check_for_zero_burst(steps, t, model[R1].as_long(), model[R2].a

        if result:
            zero_burst, found_step = result
            print(f"A burst of zeros of size {t} was found within {2**t} steps:
            print(f"It was found at step {found_step}.")
        else:
            print(f"No repeating burst of zeros of size {t} was found within {2*

        steps = 2^(t//2)

        # Check for bursts of repetition
        found_burst_info = check_for_burst_repetition(steps, t, model[R1].as_lon
```

```
        if found_burst_info:
            burst, first_step, second_step = found_burst_info
            print(f"A burst of size {t} was found that repeats within {2**(t//2)
            print(f"It was found at steps {first_step} and {second_step}.")
        else:
            print(f"No repeating burst of size {t} was found within {2**(t//2)}
    else:
        print("Properties are NOT reachable")
```

In [155…

```
solve_burst_properties(t=2)
```

```
Properties are reachable from the initial state:
R1: 349525, R2: 3355443, R3: 8388607
Generated Key: 10110111
No repeating burst of zeros of size 2 was found within 4 steps.
A burst of size 2 was found that repeats within 2 steps: 11.
It was found at steps 0 and 1.
```

Exemplo 4

Caracteristicas:

R1 == 0b00000000000000000000, equivalente a 0
R2 == 0b11001100110011001100110011, equivalente a 3355443
R3 == 0b11111111111111111111111, equivalente a 8388607
t = 4

In [156…

```
# Function to verify burst properties using Z3
def solve_burst_properties(t):
    # Create the state variables
    R1 = BitVec('R1', 19)
    R2 = BitVec('R2', 22)
    R3 = BitVec('R3', 23)

    # Initialize registers with specific pattern values
    solver = Solver()
    solver.add(R1 == 0b00000000000000000000, R2 == 0b11001100110011001100110011, R3 =

    # Check properties: verify output patterns
    solver.add(And(R1 != 0, R2 != 0, R3 != 0))

     # Check if properties are reachable
    if solver.check() == sat:
        model = solver.model()
        print('\x1b[6;30;42m' + "Properties are reachable from the initial state
        print(f"R1: {model[R1]}, R2: {model[R2]}, R3: {model[R3]}")

        # Generate and print the key
        key = generate_key(model[R1].as_long(), model[R2].as_long(), model[R3].a
        print("Generated Key:", ''.join(map(str, key)))

        # Check for bursts of zeros
        steps = 2^(t)
        result = check_for_zero_burst(steps, t, model[R1].as_long(), model[R2].a
```

```
        if result:
            zero_burst, found_step = result
            print(f"A burst of zeros of size {t} was found within {2**t} steps:
            print(f"It was found at step {found_step}.")
        else:
            print(f"No repeating burst of zeros of size {t} was found within {2*

        steps = 2^(t//2)

        # Check for bursts of repetition
        found_burst_info = check_for_burst_repetition(steps, t, model[R1].as_lon

        if found_burst_info:
            burst, first_step, second_step = found_burst_info
            print(f"A burst of size {t} was found that repeats within {2**(t//2)
            print(f"It was found at steps {first_step} and {second_step}.")
        else:
            print(f"No repeating burst of size {t} was found within {2**(t//2)}
    else:
        print('\x1b[6;30;41m' + "Properties are NOT reachable" + '\x1b[0m')
```

In [157…

```
solve_burst_properties(t=4)
```

```
Properties are NOT reachable
```