PONTIFÍCIA UNIVERSIDADE CATÓLICA
DO RIO DE JANEIRO

INF 1640

Redes de Comunicação de Dados

Relatório - Cálculo de CRC

2210872 Felipe Antelo Machado de Oliveira

Professor:
Sérgio Colcher

Rio de Janeiro
Outubro 2025

O CRC, em inglês, Cyclic Redundancy Check e, em português, Código de Redundância Cíclica é um método conhecido de detecção de erros usado no nível de enlace das redes de computadores O CRC permite verificar se os dados recebidos foram alterados durante a transmissão por ruído, interferência ou outros. O propósito deste relatório é o de realizar a tarefa proposta pelo professor e relatar os resultados adquiridos, junto a comentários e descobertas que foram observadas.

Q1) Calcule o CRC da mensagem por divisão em módulo dois utilizando o polinômio gerador X6+X4+X3+X+1 exibindo todos os passos do cálculo

R: O objetivo da primeira questão é calcular o CRC da mensagem escolhida por divisão em módulo dois utilizando o polinômio gerador específico definido por [X^6+X^4+X^3+X+1]. Para isso, foi feito um código em python e escolhida uma sequência de 32 bits aleatória.

A função principal (crc_divisao_com_passos) recebe a mensagem, adiciona seis zeros ao final (correspondentes ao grau do polinômio) e realiza uma divisão bit a bit, fazendo operações de XOR sempre que o bit atual for 1, simulando a chamada “divisão longa” em binário. Cada passo mostra o trecho de bits antes e depois da operação, permitindo acompanhar visualmente o cálculo. Ao final, os seis últimos bits do resultado formam o resto (CRC), que é anexado à mensagem original para gerar o quadro transmitido (T = D || CRC). Então, finalmente, o código que eu fiz verifica se, ao dividir novamente T pelo gerador (que depende do polinômio), o resto é zero, confirmando que o CRC foi calculado corretamente. Observe:

Sequência utilizada : 10001100011111010001110000111110
Comando para rodar a função (Input):
> python crc_divisao.py --msg 10001100011111010001110000111110
(Obviamente, após o --msg serve qualquer sequência)
Retorno do código (Output):
=== CONFIGURAÇÃO ===
Mensagem (D) : 10001100011111010001110000111110 (len=32)
Polinômio (GEN) : 1011011 (grau=6)
Dividendo inicial : 10001100011111010001110000111110000000

Passo 01 | bit 0 = 1 -> XOR em 0-6
Antes : 1000110
Div   : 1011011
Depois: 0011101
Estado: 00111010011111010001110000111110000000

Passo 02 | bit 2 = 1 -> XOR em 2-8
Antes : 1110100
Div   : 1011011
Depois: 0101111
Estado: 00010111111111010001110000111110000000

Passo 03 | bit 3 = 1 -> XOR em 3-9
Antes : 1011111
Div   : 1011011
Depois: 0000100
Estado: 00000001001111010001110000111110000000

Passo 04 | bit 7 = 1 -> XOR em 7-13
Antes : 1001111
Div   : 1011011
Depois: 0010100
Estado: 00000000010100010001110000111110000000

Passo 05 | bit 9 = 1 -> XOR em 9-15
Antes : 1010001
Div   : 1011011
Depois: 0001010
Estado: 00000000000010100001110000111110000000

Passo 06 | bit 12 = 1 -> XOR em 12-18
Antes : 1010000
Div   : 1011011
Depois: 0001011

Estado: 00000000000000010111110000111110000000

Passo 07 | bit 15 = 1 -> XOR em 15-21
Antes : 1011111
Div   : 1011011
Depois: 0000100
Estado: 00000000000000000001000000111110000000

Passo 08 | bit 19 = 1 -> XOR em 19-25
Antes : 1000000
Div   : 1011011
Depois: 0011011
Estado: 00000000000000000000011011111110000000

Passo 09 | bit 21 = 1 -> XOR em 21-27
Antes : 1101111
Div   : 1011011
Depois: 0110100
Estado: 00000000000000000000001101001110000000

Passo 10 | bit 22 = 1 -> XOR em 22-28
Antes : 1101001
Div   : 1011011
Depois: 0110010
Estado: 00000000000000000000000110010110000000

Passo 11 | bit 23 = 1 -> XOR em 23-29
Antes : 1100101
Div   : 1011011
Depois: 0111110
Estado: 00000000000000000000000011111010000000

Passo 12 | bit 24 = 1 -> XOR em 24-30
Antes : 1111101

Div   : 1011011
Depois: 0100110
Estado: 000000000000000000000000001001100000000

Passo 13 | bit 25 = 1 -> XOR em 25-31
Antes : 1001100
Div   : 1011011
Depois: 0010111
Estado: 000000000000000000000000000010111000000

Passo 14 | bit 27 = 1 -> XOR em 27-33
Antes : 1011100
Div   : 1011011
Depois: 0000111
Estado: 000000000000000000000000000000001110000

Passo 15 | bit 31 = 1 -> XOR em 31-37
Antes : 1110000
Div   : 1011011
Depois: 0101011
Estado: 000000000000000000000000000000000101011

=== RESULTADOS ===
CRC/FCS (r=6)   : 101011
Transmitido (T=D||C): 10001100011111010001110000111110101011

Q2) Mostre o LFSR simplificado para o polinômio gerador da questão 2.

R: A questão pede para mostrar o Registros de Deslocamento com Realimentação Linear (Linear Feedback Shift Registers, vulgo, LFSRs). Dado que o polinômio gerador da questão foi X^6+X^4+X^3+X+1, teremos então que os bits representativos do gerador serão “1011011”. Abaixo eu fiz no tablet uma esquemática do LFSR simplificado. Junto da esquemática há observações importantes para entender o raciocínio. Observe:

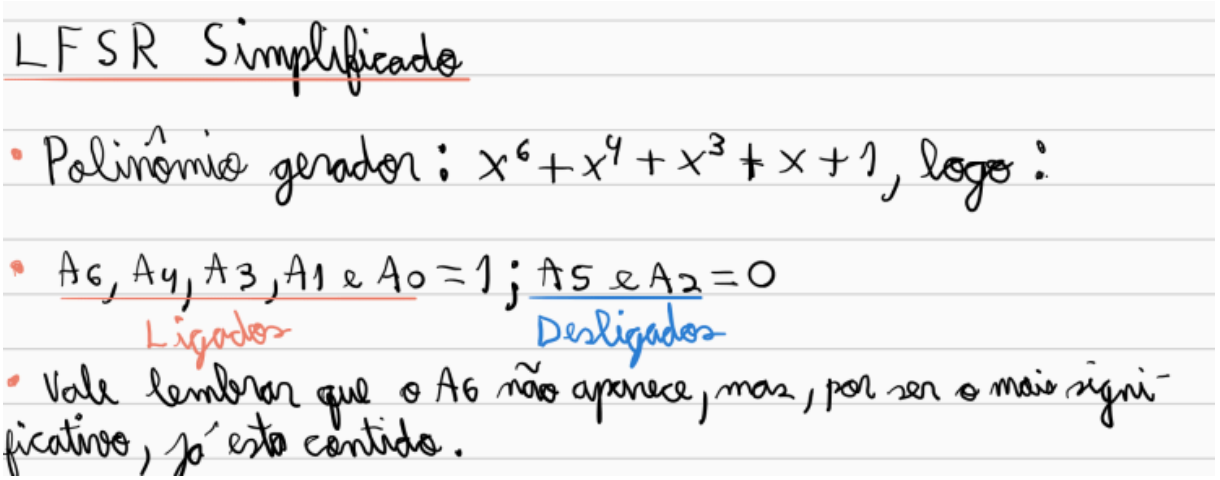


Imagem 1 - Descrição lógica LFSR

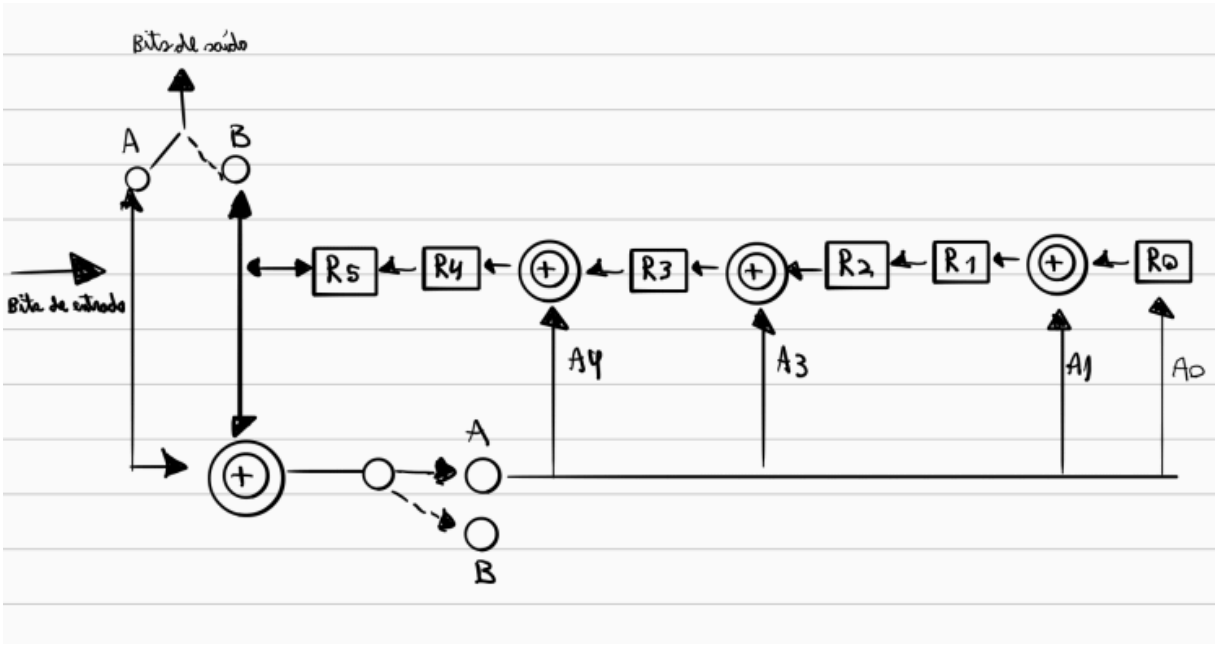


Imagem 2 - Montagem visual LFSR

Q3) Faça um quadro mostrando a evolução do funcionamento do LFSR até chegar a resposta. Compare a resposta final com a obtida na questão 1.

R: A questão pede, agora, gradualmente, para mostrar o passo a passo da evolução do LFSR até chegar na resposta. Devemos observar que o resultado deve ser exatamente igual à resposta alcançada na questão 1, do contrário, algo errado foi feito. Novamente, fiz uma

esquemática no tablet para visualizar. Observe:

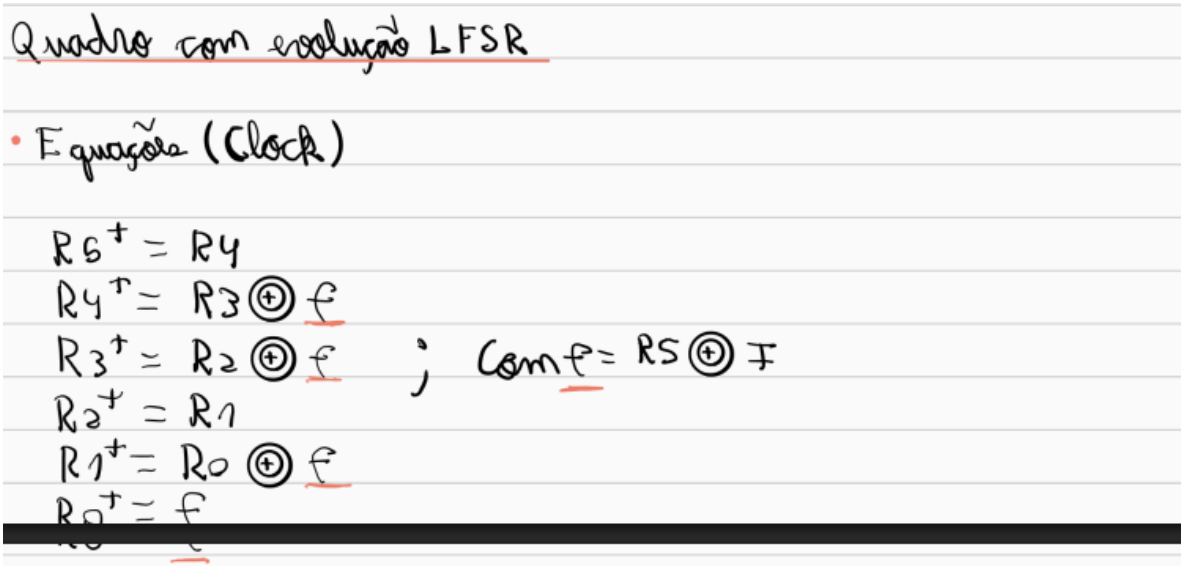
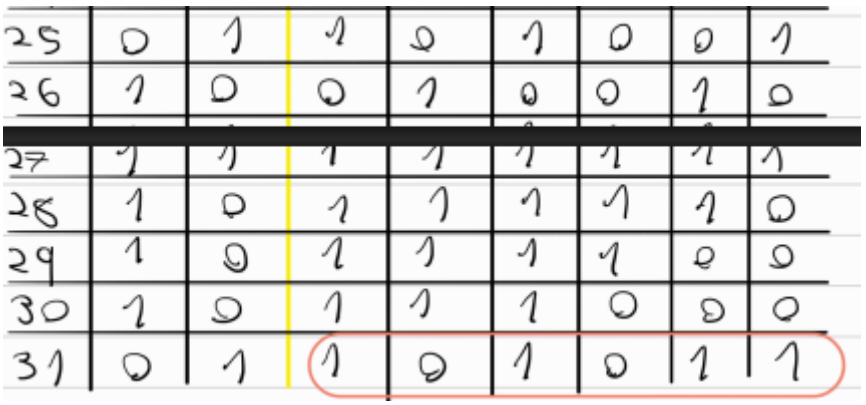


Imagem 3 - Equações para LFSR Clock

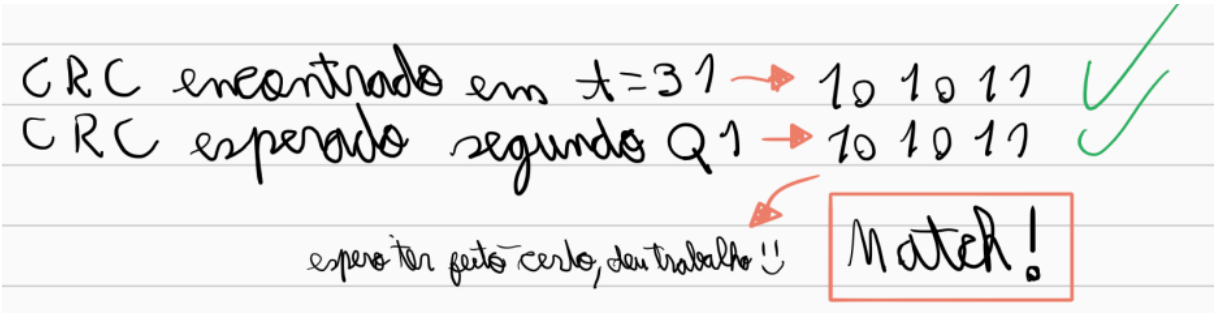
Obs: f = R5(t) + I(t)

Tabela gradual:

t	I	f	R_5	R_4	R_3	R_2	R_1	R_0
0	1	1	0	1	1	0	1	1
1	0	0	1	1	0	1	1	0
2	0	1	1	1	0	1	1	1
3	0	1	1	1	0	1	0	1
4	1	0	1	0	1	0	1	0
5	1	0	0	1	0	1	0	0
6	0	0	1	0	1	0	0	0
7	0	1	0	0	1	0	1	1
8	0	0	0	1	0	1	1	0
9	1	1	1	1	0	1	1	1
10	1	0	1	0	1	1	1	0
11	1	0	0	1	1	1	0	0
12	1	1	1	0	0	0	1	1
13	1	0	0	0	0	1	1	0
14	0	0	0	0	1	1	0	0
15	1	1	0	0	0	0	1	1
16	0	0	0	0	0	1	1	0
17	0	0	0	0	1	1	0	0
18	0	0	0	1	1	0	0	0
19	1	1	1	0	1	0	1	1
20	1	0	0	1	0	1	1	0
21	1	1	1	1	0	1	1	0
22	0	1	1	1	0	1	0	1
23	0	1	1	1	0	0	0	1
24	0	1	1	1	1	0	0	1



25	0	1	1	0	1	0	0	1
26	1	0	0	1	0	0	1	0
27	1	1	1	1	1	1	1	1
28	1	0	1	1	1	1	1	0
29	1	0	1	1	1	1	0	0
30	1	0	1	1	1	0	0	0
31	0	1	1	0	1	0	1	1



Imagens 4, 5 e 6 - Quadro LFSR + Observação final

Conclusão: Aplicando o LFSR , inicializado em 0 e alimentado com a sequência escolhida de: 10001100011111010001110000111110, o estado ao final da fase A (t=31) é 101011, que coincide com o CRC da questão 1, comprovando o sucesso do exercício.

Despedida: Espero que tenha ficado tudo certo e o relatório tenha ficado adequado. Como o EAD não permite anexar arquivos que não fosse .pdf, coloquei o código completo utilizado na questão 1 abaixo. Espero que esteja tudo bem. Caso necessário, pode me solicitar qualquer parte que estiver faltando no meu e-mail: felipe.antelo.machado@gmail.com
Abraços!

Código Completo (Questão 1):

```
#Aluno: Felipe Antelo M. de Oliveira
#Matrícula: 2210872
#Professor: Sérgio Colcher
#Polinômio gerador do exercício: P(x) = X^6 + X^4 + X^3 + X + 1  ->
bits "1011011" (grau 6)

from typing import List, Tuple

GEN = "1011011"
R = len(GEN) - 1 #Feito para garantir grau 6

def _valida_bits(s: str) -> None:
    if not s or any(c not in "01" for c in s):
        raise ValueError("Forneça só 0/1.")

def _xor_inplace(buf: List[str], pat: str, i: int) -> None:
    for j, b in enumerate(pat):
        buf[i+j] = '0' if buf[i+j] == b else '1'

def _print_passo(k: int, i: int, antes: str, gen: str, depois: str,
total: str) -> None:
    print(f"Passo {k:02d} | bit {i} = 1 -> XOR em {i}-{i+len(gen)-1}")
    print(f"Antes : {antes}")
    print(f"Div   : {gen}")
    print(f"Depois: {depois}")
    print(f"Estado: {total}\n")

def crc_divisao_com_passos(msg: str, gen: str = GEN) -> Tuple[str,
str]:
    _valida_bits(msg)
    r = len(gen) - 1
    dividendo = msg + "0"*r
    a = list(dividendo)

    print("=== CONFIGURAÇÃO ===")
    print(f"Mensagem (D)        : {msg}  (len={len(msg)})")
    print(f"Polinômio (GEN)     : {gen}  (grau={r})")
    print(f"Dividendo inicial   : {dividendo}\n")
```

```
    k = 0
    for i in range(0, len(a) - len(gen) + 1):
        if a[i] == '1':
            k += 1
            janela_antes = ''.join(a[i:i+len(gen)])
            _xor_inplace(a, gen, i)
            janela_depois = ''.join(a[i:i+len(gen)])
            _print_passo(k, i, janela_antes, gen, janela_depois,
''.join(a))
        # se a[i]=='0': nenhum passo (como na divisão longa)

    crc = ''.join(a[-r:]) if r>0 else ""
    T = msg + crc
    print("=== RESULTADOS ===")
    print(f"CRC/FCS (r={r})     : {crc}")
    print(f"Transmitido (T=D||C): {T}")

    # verificação: T ÷ GEN deve ter resto zero
    ok = verifica_resto_zero(T, gen)
    print(f"\nVerificação (T ÷ GEN): resto {'0'*r} -> {'OK' if ok else
'ERRO'}")
    return crc, T

def verifica_resto_zero(bits: str, gen: str = GEN) -> bool:
    a = list(bits)
    m = len(gen)
    for i in range(0, len(a) - m + 1):
        if a[i] == '1':
            _xor_inplace(a, gen, i)
    resto = a[-(m-1):] if m > 1 else []
    return all(c == '0' for c in resto)

if __name__ == "__main__":
    import argparse
    p = argparse.ArgumentParser(description="CRC (divisão módulo-2) com
passos. GEN=1011011.")
    p.add_argument("--msg", required=True, help="Mensagem binária (ex.:
32 bits).")
    args = p.parse_args()
```

```
    crc_divisao_com_passos(args.msg, GEN)
```