Arquitetura e Organização de Computadores

UFRR

APRESENTANDO O PROJETO FINAL

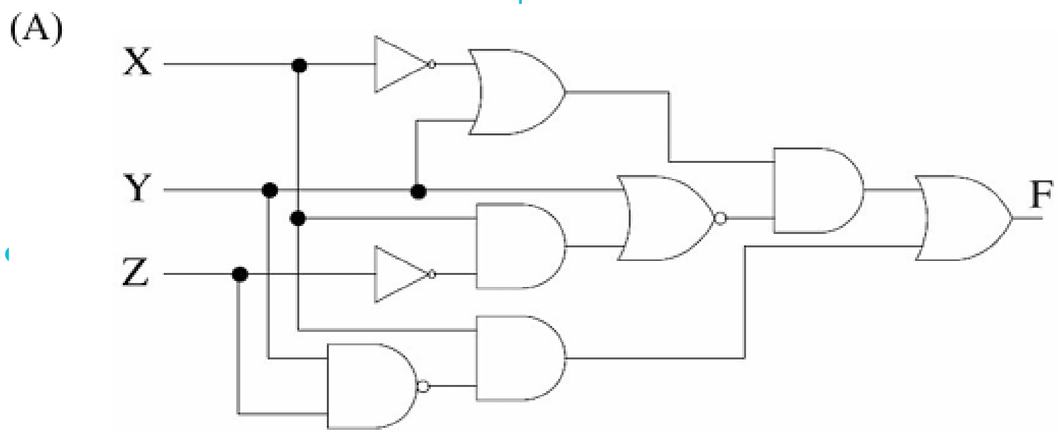
TASK 02

José Victor Rocha
Luiz Gustavo Cavalcante
Guilherme Miranda

ELABORAÇÃO

O código fora elaborado no Google Colab, utilizando a linguagem Python e separado em blocos

CIRCUITO A



Circuito representado pela seguinte equação booleana:
F=Or(And(¬X,Y),And(¬Y,Z),And(X,¬Y),And(X,¬Z)

Após implementação em código é notável que o circuito possui um F falso quando X == false Y == false Z == false
Entretanto também é perceptível que os circuitos não são equivalentes, ou seja não é possível remover a redundância.

Verificação de equivalência

```
[ ]  from z3 import *
     # Definir variáveis booleanas
     X, Y, Z = Bools('X Y Z')
     # Circuito original (com redundância)
     F_original = Or(And(Not(X), Y), Not(Or(Y, And(X, Not(Z)))), And(X, Not(And(Y, Z))))
     # Circuito simplificado (sem redundância)
     F_simplificado = Or(Not(Y), And(X, Not(Z)))
     # Solver para verificar equivalência
     solver = Solver()
     # Adicionar condição de inequivalência (se forem diferentes, há um problema)
     solver.add(F_original != F_simplificado)
     # Verificar
     if solver.check() == sat:
       print("Os circuitos NÃO são equivalentes. A redundância não pode ser removida.")
     else:
       print("Os circuitos são equivalentes. A redundância pode ser removida.")
```

```
Os circuitos NÃO são equivalentes. A redundância não pode ser removida.
```

CIRCUITO B

SOMADOR 8 BITS

A	B	C	Xor(A7, B7, C7)
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

Expressões Booleanas:
Or(And(A[i-1], B[i-1]), And(C[i-1], Xor(A[i-1], B[i-1])))
Or(And(B, C), And(A, C), And(A, B))
Xor(A[7], B[7], C[7])

Após implementação em código fora notada como era possível a simplificação do circuito cortando o xor, após análise é confirmado que ambos os circuitos são equivalentes.

CIRCUITO C

UNIDADE DE CONTROLE

Para o código deste circuito um método diferente fora utilizado, memória se tornou uma variável, os estados da unidade de controle se tornaram booleanas e fora criado uma expressão para cada bit de saída.

A partir desta analise foi possível concluir que o circuito não possui elementos redundantes.

Estado	Descrição	Função/Responsabilidade
S0	**Busca da instrução (Fetch Cycle)**	Inicializa o processo de busca da instrução. O **PC (Program Counter)** é atualizado para buscar a próxima instrução na memória e ela é carregada no **IR (Instruction Register)**.
S1	**Decodificação da instrução (Decode)**	O processador lê os registradores de origem, interpreta a instrução e prepara os operandos para execução. Aqui, os valores dos registradores são lidos e a operação é decodificada.
S2	**Execução (Execute)**	A operação da **ALU** (Aritmética e Lógica) é executada neste estágio, seja uma soma, subtração, ou operação de comparação (como no caso de **BEQ** ou **BNE**). Pode também ser o estágio onde são calculados os endereços de memória para operações de load/store.
S3	**Acesso à Memória (Memory Access)**	Se a instrução envolver acesso à memória (como **LW** ou **SW**), neste estágio o processador lê ou escreve na memória. Para **LW**, os dados são lidos da memória; para **SW**, os dados são gravados.
S4	**Escrita no Registrador (Write Back)**	O resultado da execução da instrução (seja de uma operação ALU ou de leitura de memória) é escrito de volta no registrador de destino. Esse é o estágio final da execução de uma instrução.
S5	**Leitura de Memória para Registrador (Load Word)**	Durante este estágio, uma instrução de **LW** (Load Word) carrega dados da memória para um registrador.
S6	**Controle de Condição de Branch (Branch Control)**	Para instruções de ramificação, como **BEQ** ou **BNE**, o processador verifica a condição de branch. Se a condição for verdadeira, o **PC** é atualizado para o endereço de destino da ramificação.

CIRCUITO D

Importando o circuito para Z3 e realizando verificações básicas (f é falso para A==false, D==false, B==true, C==true) é notável que existem sim redundâncias e que o circuito poderia ser simplificado para a seguinte expressão:(¬A·¬C·D)+(¬A·B·C)+(A·C·D)+(A·B·¬C)

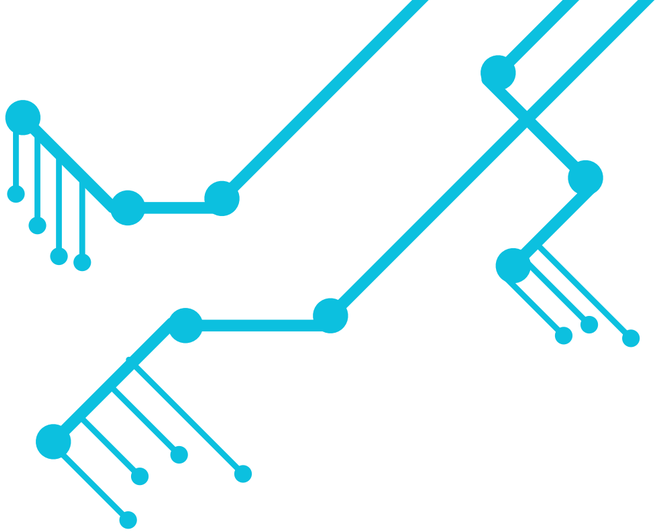
$$(\neg A \cdot \neg C \cdot D) + (\neg A \cdot B \cdot C) + (A \cdot C \cdot D) + (A \cdot B \cdot \neg C)$$

CIRCUITO E

ULA 8 BITS

Para cada operação da ULA fora criada uma expressão diferente, então testando no código é possível notar que não há como simplificar o circuito, pois não há redundâncias.

```python
from z3 import *

# Definição das variáveis de entrada (8 bits)
A = BitVec('A', 8)
B = BitVec('B', 8)
Op = BitVec('Op', 3)  # Código de operação

# Definição das operações
subtracao = A + (~B + 1)  # A - B usando complemento de dois
xor_op = A ^ B  # XOR
nand_op = ~(A & B)  # NAND
nor_op = ~(A | B)  # NOR
shift_left_2 = A << 2  # Shift à esquerda de 2 bits

# Seleção da operação com base no código Op
Result = If(Op == 0, subtracao,
         If(Op == 1, xor_op,
         If(Op == 2, nand_op,
         If(Op == 3, nor_op,
         If(Op == 4, shift_left_2, BitVecVal(0, 8))))))  # Retorna 0 se Op for inválido

# Lista de testes com suas respectivas operações e valores
testes = [
    ("Subtração", subtracao, 0, 20, 5),  # A=20, B=5, Op=0 (Subtração)
    ("XOR", xor_op, 1, 0b10101010, 0b11001100),  # A=170, B=204, Op=1 (XOR)
    ("NAND", nand_op, 2, 0b00001111, 0b11110000),  # A=15, B=240, Op=2 (NAND)
    ("NOR", nor_op, 3, 0b10101010, 0b01010101),  # A=170, B=85, Op=3 (NOR)
    ("Shift Left 2", shift_left_2, 4, 0b00001111, 0)  # A=15, Op=4 (Shift Left 2)
]

# Testa cada operação no solver
for nome, operacao, op_code, a_val, b_val in testes:
    solver = Solver()
    solver.add(Not(Result == operacao))  # Verificar se a operação está incorreta
    solver.add(A == a_val, B == b_val, Op == op_code)

    print(f"🔍 Verificando {nome}:")
    if solver.check() == sat:
        print("    ❌ A especificação falhou.")
        print("   Modelo encontrado:", solver.model())
    else:
        print("    ✅ A especificação foi atendida.")
    print("-" * 40)
```



CIRCUITO F

Para este circuito cada estado foi representado com diferente expressões booleanas, flags de controle de instrução foram representadas por expressões, então essas representações foram adicionadas ao solver para verificar sua consistência, sendo concluído que o circuito não possui elementos redundantes