



UFRR

**PODER EXECUTIVO
MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DE RORAIMA
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO**

ARQUITETURA E ORGANIZAÇÃO DE COMPUTADORES

**RELATÓRIO TASK 02: ANÁLISE E VERIFICAÇÃO DE CIRCUITOS LÓGICOS COM FÓRMULAS
BOOLEANAS**

ALUNOS:

**Luiz Gustavo Dall'Agnol Cavalcante – 2021000632
José Victor Rocha de Alencar – 2021000570
Guilherme Miranda de Araújo - 2021019643**

**Março de 2025
Boa Vista/Roraima**



UFRR

**PODER EXECUTIVO
MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DE RORAIMA
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO**

ARQUITETURA E ORGANIZAÇÃO DE COMPUTADORES

**RELATÓRIO TASK 02: ANÁLISE E VERIFICAÇÃO DE CIRCUITOS LÓGICOS COM FÓRMULAS
BOOLEANAS**

**Março de 2025
Boa Vista/Roraima**

Resumo

Este trabalho apresenta a análise e verificação dos circuitos digitais propostos na Task 02, que incluem somadores, unidades de controle, unidades lógicas e aritméticas, além de expressões booleanas representativas de circuitos lógicos. A implementação foi realizada utilizando a linguagem Python no ambiente Google Colab, empregando o solver Z3 para modelagem e validação lógica. Os resultados obtidos a partir da verificação das especificações, como equivalência lógica, redundância de componentes e validade das saídas, serão descritos a seguir.

Link para os códigos

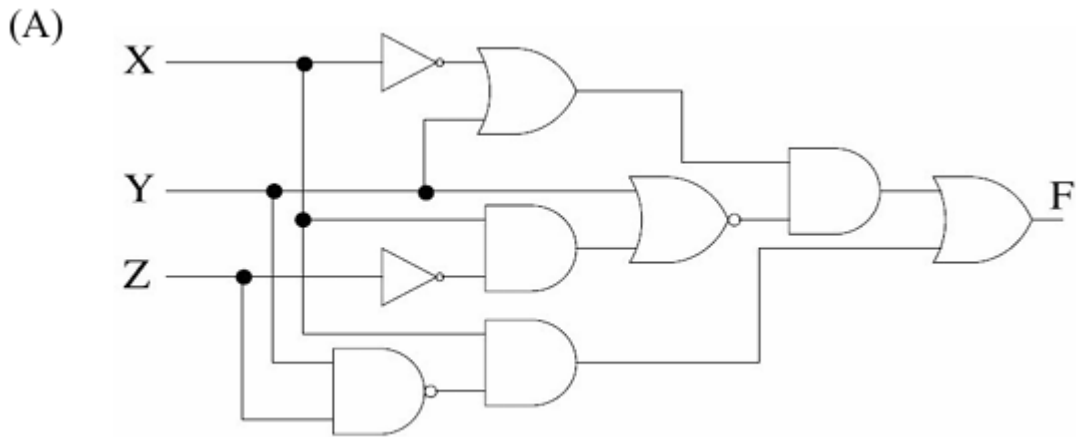
https://colab.research.google.com/drive/17MOU-eXkeMPZJ_OhD37D9-3FMdxsM2aq?usp=sharing#scrollTo=ImsdeKIoZIXJ

Conteúdo

1.	Circuitos: Validação de saída, Redundância de componentes, Formas simplificadas	5
1.1	Circuito A	5
1.2	Circuito B	5
1.3	Circuito C	6
<u>1.4</u>	Circuito D	6
1.5	Circuito E	8
<u>1.6</u>	Circuito F	12

1. Circuitos: Validação de saída, Redundância de componentes, Formas simplificadas

1.1 Circuito A



Expressão booleana : $F = \text{Or}(\text{And}(\neg X, Y), \text{And}(\neg Y, Z), \text{And}(X, \neg Y), \text{And}(X, \neg Z))$

Validação de saída:

```
# Especificação: A saída deve ser verdadeira se X == False, Y == False, Z == False
solver.add(Not(F)) # Verificar se F é falso para a condição especificada
solver.add(X == False, Y == False, Z == False)
# Resultado da verificação
if solver.check() == sat:
    print("A especificação falhou.")
else:
    print("A especificação foi atendida.")
```

A especificação foi atendida.

Forma simplificada:

Após a simplificação

```
from z3 import *
# Definição das variáveis booleanas
X, Y, Z = Bools('X Y Z')

# Expressão convertida para Z3
F = Or(Not(Y), And(X, Not(Z)))
# Solver para verificar a especificação
solver = Solver()

[ ] # Especificação: A saída deve ser verdadeira se X == False, Y == False, Z == False
solver.add(Not(F)) # Verificar se F é falso para a condição especificada
solver.add(X == False, Y == False, Z == False)
# Resultado da verificação
if solver.check() == sat:
    print("A especificação falhou.")
else:
    print("A especificação foi atendida.")
```

A especificação foi atendida.

Verificação de equivalência:

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.

1.2 Circuito B

```
from z3 import *

# Definir variáveis de 8 bits para A e B
A = [Bool(f'A{i}') for i in range(8)]
B = [Bool(f'B{i}') for i in range(8)]
C = [False] * 8 # Inicializar os carries

# Definir carry-in inicial
C[0] = False # Carry-in inicial é 0

# Definir a equação dos carries
for i in range(1, 8):
    C[i] = Or(And(A[i-1], B[i-1]), And(C[i-1], Xor(A[i-1], B[i-1])))

# Equação do bit mais significativo (MSB) da soma
S_7 = Xor(A[7], B[7], C[7])

# Criar o solver
solver = Solver()
```

```
[ ] # Testar uma entrada específica (exemplo: A = 10000000, B = 10000000)
solver.add(A[7] == True, A[6] == False, A[5] == False, A[4] == False, A[3] == False, A[2] == False, A[1] == False, A[0] == False)
solver.add(B[7] == True, B[6] == False, B[5] == False, B[4] == False, B[3] == False, B[2] == False, B[1] == False, B[0] == False)

# Verificar se S_7 está correto
solver.add(Not(S_7)) # Verificamos se o resultado esperado não acontece

# Checar a especificação
if solver.check() == sat:
    print("A especificação falhou.") # O MSB não se comportou corretamente
else:
    print("A especificação foi atendida.") # O MSB está correto
```

➡ A especificação foi atendida.

```
▶ # Definir variáveis booleanas
A = Bool('A')
B = Bool('B')
C = Bool('C')

# Expressão:  $A \cdot B + (A \oplus B) \cdot C$ 
F = Or(And(A, B), And(Xor(A, B), C))

# Criar solver e adicionar restrições (se necessário)
solver = Solver()
solver.add(F == True) # Exemplo: testar quando F é verdadeiro

# Verificar se a expressão é satisfazível
if solver.check() == sat:
    print("A expressão é satisfazível.")
else:
    print("A expressão não é satisfazível.")
```

➡ A expressão é satisfazível.

```

# Definir variáveis booleanas
A = Bool('A')
B = Bool('B')
C = Bool('C')

# Expressão:  $A.B + (A \oplus B).C$ 
F = Or(And(B, C), And(A, C), And(A, B))

# Criar solver e adicionar restrições (se necessário)
solver = Solver()
solver.add(F == True) # Exemplo: testar quando F é verdadeiro

# Verificar se a expressão é satisfazível
if solver.check() == sat:
    print("A expressão é satisfazível.")
else:
    print("A expressão não é satisfazível.")

```

→ A expressão é satisfazível.

```

[ ] from z3 import *
# Definir variáveis booleanas
A = Bool('A')
B = Bool('B')
C = Bool('C')
# Circuito original (com redundância)
F_original = Or(And(A, B), And(Xor(A, B), C))
# Circuito simplificado (sem redundância)
F_simplificado = Or(And(B, C), And(A, C), And(A, B))
# 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 são equivalentes. A redundância pode ser removida.

1.3 Circuito C

```
[ ] from z3 import *

# Definição das variáveis booleanas
S0, S1, S2, S3, S4, S5, S6 = Bools('S0 S1 S2 S3 S4 S5 S6')
JUMP, BEQ, BNE, TipoR, SW, LW = Bools('JUMP BEQ BNE TipoR SW LW')

# Definição das expressões booleanas
PCWrite = Or(S0, And(S6, JUMP))
PCWriteCond = And(S6, Or(BEQ, BNE))
IorD = Or(S3, S5)
MemRead = Or(S0, S3)
MemWrite = And(S3, SW)
MemtoReg = S5
IRWrite = S0
ALUSrcA = Or(S1, S2, S3)
ALUSrcB1 = Or(S2, S3)
ALUSrcB0 = S1
ALUOp1 = And(S2, TipoR)
ALUOp0 = And(S6, Or(BEQ, BNE))
RegWrite = Or(And(S4, TipoR), And(S5, LW))
RegDst = And(S4, TipoR)

# Lista das expressões e seus nomes
expressoes = {
    "PCWrite": PCWrite,
    "PCWriteCond": PCWriteCond,
    "IorD": IorD,
    "MemRead": MemRead,
    "MemWrite": MemWrite,
    "MemtoReg": MemtoReg,
    "IRWrite": IRWrite,
    "ALUSrcA": ALUSrcA,
    "ALUSrcB1": ALUSrcB1,
    "ALUSrcB0": ALUSrcB0,
    "ALUOp1": ALUOp1,
    "ALUOp0": ALUOp0,
    "RegWrite": RegWrite,
    "RegDst": RegDst
}
```

```
# Verificação de satisfatibilidade para cada expressão
for nome, expressao in expressoes.items():
    solver = Solver()
    solver.add(expressao) # Adiciona a expressão ao solver

    if solver.check() == sat:
        modelo = solver.model()
        print(f"{nome}: SATISFATÍVEL")
        print("    Exemplo de atribuições que satisfazem:", {v: modelo[v] for v in modelo})
    else:
        print(f"{nome}: INSATISFATÍVEL")
print("-" * 40)
```

```
PCWrite: SATISFATÍVEL
    Exemplo de atribuições que satisfazem: {S0: False, S6: True, JUMP: True}
-----
PCWriteCond: SATISFATÍVEL
    Exemplo de atribuições que satisfazem: {BEQ: True, S6: True, BNE: False}
-----
IorD: SATISFATÍVEL
    Exemplo de atribuições que satisfazem: {S3: True, S5: False}
-----
MemRead: SATISFATÍVEL
    Exemplo de atribuições que satisfazem: {S0: True, S3: False}
-----
MemWrite: SATISFATÍVEL
    Exemplo de atribuições que satisfazem: {S3: True, SW: True}
-----
MemtoReg: SATISFATÍVEL
    Exemplo de atribuições que satisfazem: {S5: True}
-----
IRWrite: SATISFATÍVEL
    Exemplo de atribuições que satisfazem: {S0: True}
-----
ALUSrcA: SATISFATÍVEL
    Exemplo de atribuições que satisfazem: {S2: False, S1: True, S3: False}
-----
ALUSrcB1: SATISFATÍVEL
    Exemplo de atribuições que satisfazem: {S2: True, S3: False}
-----
ALUSrcB0: SATISFATÍVEL
    Exemplo de atribuições que satisfazem: {S1: True}
-----
ALUOp1: SATISFATÍVEL
    Exemplo de atribuições que satisfazem: {S2: True, TipoR: True}
-----
ALUOp0: SATISFATÍVEL
    Exemplo de atribuições que satisfazem: {BEQ: True, S6: True, BNE: False}
-----
RegWrite: SATISFATÍVEL
    Exemplo de atribuições que satisfazem: {TipoR: True, S4: True, S5: False, LW: False}
-----
RegDst: SATISFATÍVEL
    Exemplo de atribuições que satisfazem: {TipoR: True, S4: True}
-----
```

Circuito não possui elementos redundantes.

1.4 Circuito D

```
from z3 import *


# Definição das variáveis booleanas
A, B, C, D = Bools('A B C D')

# Expressão booleana em Z3
expr = Or(And(A, B, C, D),
          And(A, B, Not(C), D),
          And(A, B, Not(C), Not(D)),
          And(A, Not(B), C, D),
          And(Not(A), B, C, D),
          And(Not(A), B, C, Not(D)),
          And(Not(A), B, Not(C), D),
          And(Not(A), Not(B), Not(C), D))

# Criando um solver para verificar a expressão
solver.add(Not(expr)) # Verificar se F é falso para a condição especificada
solver.add(A == False, D == False, B == True, C == True)
# Resultado da verificação
if solver.check() == sat:
    print("A especificação falhou.")
else:
    print("A especificação foi atendida.")
```

⇒ A especificação foi atendida.


Após a simplificação

```
 from z3 import *

# Definição das variáveis booleanas
A, B, C, D = Bools('A B C D')

# Expressão booleana em Z3
expr = Or(And(Not(A), Not(C), D),
          And(Not(A), B, C),
          And(A, C, D),
          And(A, B, Not(C)))

# Criando um solver para verificar a expressão
solver.add(Not(expr)) # Verificar se F é falso para a condição especificada
solver.add(A == False, D == False, B == True, C == True)
# Resultado da verificação
if solver.check() == sat:
    print("A especificação falhou.")
else:
    print("A especificação foi atendida.")
```

 A especificação foi atendida.

```

from z3 import *

# Definir variáveis booleanas
A, B, C, D = Bools('A B C D')
# Circuito original (com redundância)
F_original = Or(And(A, B, C, D),
                And(A, B, Not(C), D),
                And(A, B, Not(C), Not(D)),
                And(A, Not(B), C, D),
                And(Not(A), B, C, D),
                And(Not(A), B, C, Not(D)),
                And(Not(A), B, Not(C), D),
                And(Not(A), Not(B), Not(C), D))
# Circuito simplificado (sem redundância)
F_simplificado = Or(And(Not(A), Not(C), D),
                    And(Not(A), B, C),
                    And(A, C, D),
                    And(A, B, Not(C)))
# 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 são equivalentes. A redundância pode ser removida.

1.5 Circuito E

```

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)
```

```
🔍 Verificando Subtração:
✅ A especificação foi atendida.
-----
🔍 Verificando XOR:
✅ A especificação foi atendida.
-----
🔍 Verificando NAND:
✅ A especificação foi atendida.
-----
🔍 Verificando NOR:
✅ A especificação foi atendida.
-----
🔍 Verificando Shift Left 2:
✅ A especificação foi atendida.
-----

Circuito não possui elementos redundantes.
```

1.6 Circuito F