Verificação Formal de Circuitos Digitais utilizando Lógica Booleana e o Solver Z3

Daniel Marchioro[1], Inácio L. S. Viana[1], Sarah C. A. Pereira[1]

[1]Universidade Federal de Roraima
Departamento de Ciência da Computação – Boa Vista, RR – Brasil

m4chiorouniv@gmail.com, inacioviana.iv@gmail.com,
sarahscarolinec@gmail.com

Abstract. *This paper presents a formal verification methodology for digital circuits using Boolean logic and the Z3 theorem prover. Six circuits were analyzed: (A) a combinational logic circuit, (B) an 8-bit full adder, (C) a finite state machine control unit for a multiclock MIPS processor, (D) a complex Boolean expression, (E) an 8-bit Arithmetic Logic Unit with multiple functions, and (F) the execution flow of an R-type instruction in MIPS. Each circuit was converted into Boolean formulas, which were then verified using Z3 for output correctness, component redundancy, and logical equivalence. The results demonstrate the effectiveness of automated verification in identifying redundancies and ensuring circuit correctness, with potential applications in digital design and optimization.*

Resumo. *Este artigo apresenta uma metodologia de verificação formal de circuitos digitais utilizando lógica booleana e o prover de teoremas Z3. Seis circuitos foram analisados: (A) um circuito combinacional, (B) um somador completo de 8 bits, (C) uma unidade de controle por máquina de estados finitos para um processador MIPS multiciclo, (D) uma expressão booleana complexa, (E) uma Unidade Lógica e Aritmética de 8 bits com múltiplas funções, e (F) o fluxo de execução de uma instrução tipo R no MIPS. Cada circuito foi convertido em fórmulas booleanas, que foram então verificadas usando Z3 quanto à correção da saída, redundância de componentes e equivalência lógica. Os resultados demonstram a eficácia da verificação automatizada na identificação de redundâncias e garantia da correção dos circuitos, com aplicações potenciais em projeto e otimização digital.*

1. Introdução

Projetar sistemas computacionais eficientes exige a eliminação de redundâncias em suas bases lógicas. Neste contexto, exploramos como simplificar expressões booleanas de forma estruturada utilizando Python, aplicando bibliotecas como SymPy para a manipulação dos termos e o Z3 para certificar que a lógica original foi preservada. Ao reduzir a complexidade dessas expressões, viabilizamos a criação de circuitos digitais de alto desempenho, mantendo a total fidelidade às funções lógicas planejadas.

2. Metodologia

O processo de modelagem, análise comportamental, simplificação e comparação essencialmente seguem os mesmos passos para todos os circuitos apresentados neste trabalho; primeiramente o circuito é analisado, e dele é retirado sua fórmula booleana, esta que, é alimentada a um programa escrito em python, utilizando a biblioteca Z3 para sua simulação. O mesmo circuito então, passa por outro programa que a partir das ferramentas de simplificações lógicas oferecidas pela biblioteca SymPy, verifica a

fórmula booleana utilizada e a simplifica se possível. O novo circuito passa então pelo mesmo processo que o original, para validar se as saídas atendem o comportamento correto do circuito. Por fim, as duas fórmulas são postas em um mesmo programa utilizando o Z3 novamente, para comparar se elas são equivalentes, sendo essa a etapa de confirmação que o circuito de fato pode ser simplificado sem mais problemas.

2.1 Circuitos Analisados

Ao todo foram analisados seis circuitos digitais:

- **(A) Um circuito combinacional;**

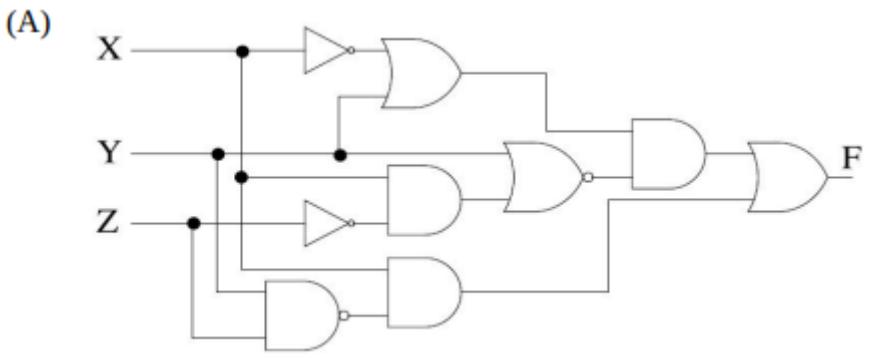

Figura 1. Circuito combinacional (A)

- **(B) Um somador de 8 bits;**

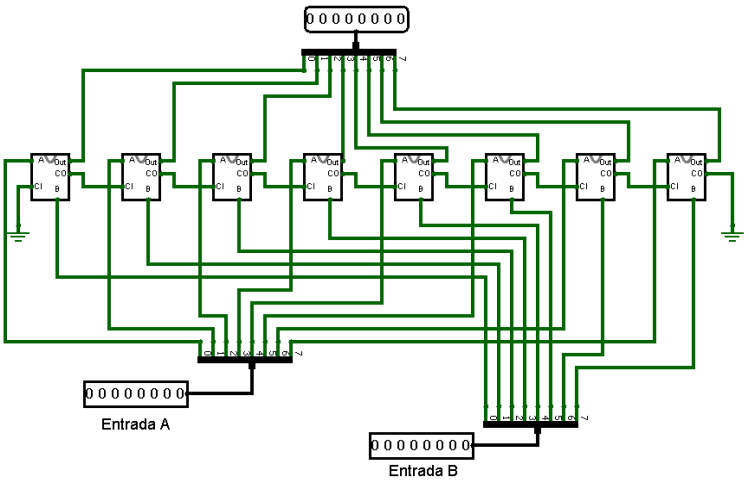

Figura 2. Representação de um Somador de 8 Bits

- **(C) Uma unidade de controle (codificada em uma Máquina de Estados Finitos Completa) do processador multiciclo MIPS;**

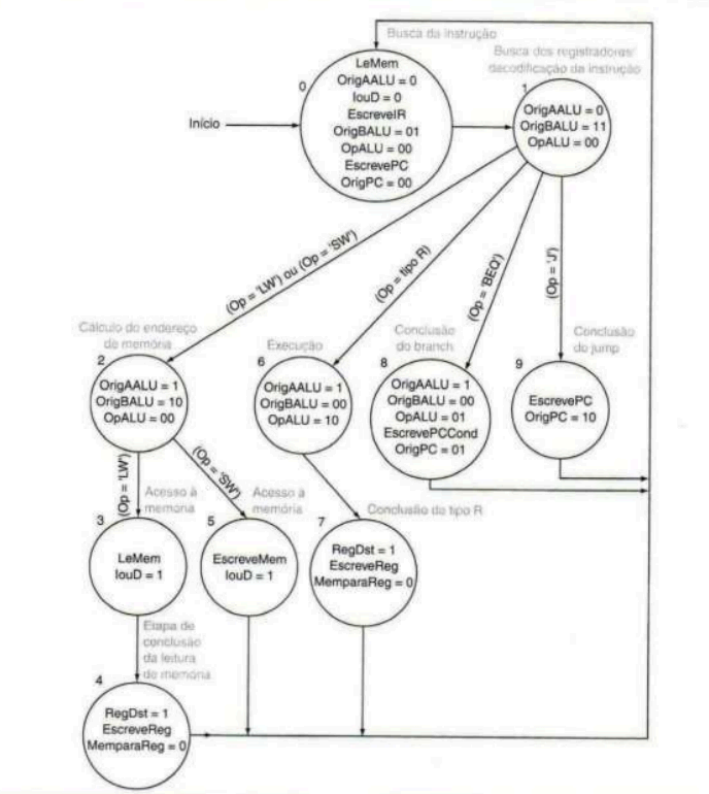

Figura 3. Máquina de Estados Finitos Completo

- **(D) Outro circuito combinacional;**

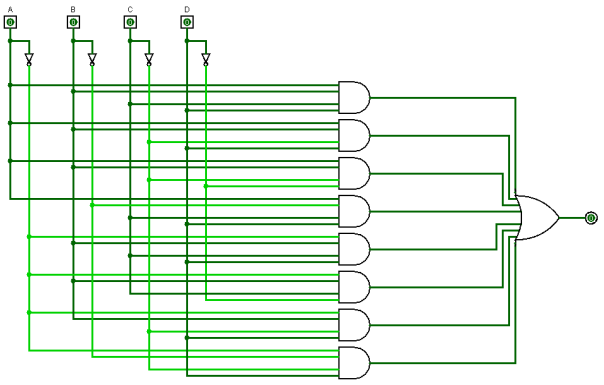

Figura 4. Circuito combinacional (D)

- **(E) Uma Unidade Lógica e Aritmética (ULA) de 8 bits com as funções de subtração, XOR, NAND, NOR e shift de 2 bits à esquerda;**

- **(F) O Fluxo de execução de uma instrução do tipo R no MIPS de 32 bits.**

Figura 5. Fluxo de uma instrução do tipo R

2.2 Conversão para Fórmulas Booleanas

Analisando cada circuito, obtemos as seguintes expressões booleanas:

- **(A)** $F = ((\neg X \wedge Y) \wedge \neg(Y \vee (X \wedge \neg Z))) \vee (X \vee \neg(Y \wedge Z))$;
- **(B)** Para um circuito somador completo, temos dois sinais de saída: o resultado da soma e o Carry Out:

  $S = A \oplus B \oplus \text{Carry In}$

  $\text{Carry Out} = (A \wedge B) \vee (\text{Carry In} \wedge (A \vee B))$

  A modelagem de um circuito somador de 8 bits segue uma lógica em cascata, onde o Carry In sempre vai receber o Carry Out do somador anterior:

  $S(n) = A(n) \oplus B(n) \oplus \text{Carry In}(n)$

Carry Out(n) = (A(n) ∧ B(n)) ∨ (Carry In(n) ∧ (A(n) ∨ B(n)));

- **(C)** Os sinais de saída são definidos como as opções de controle oferecidas pelo processador MIPS, são eles:

  Jump = ¬S2 ∧ S1 ∧ ¬S0 ∧ ¬Op5 ∧ ¬Op4 ∧ ¬Op3 ∧ ¬Op2 ∧ Op1 ∧ Op0;

  ALUOp0 = ¬S2 ∧ S1 ∧ ¬S0 ∧ ¬Op5 ∧ ¬Op4 ∧ ¬Op3 ∧ ¬Op2 ∧ Op1 ∧ ¬Op0;

  ALUOp1 = ¬S2 ∧ S1 ∧ ¬S0 ∧ ¬Op5 ∧ ¬Op4 ∧ ¬Op3 ∧ ¬Op2 ∧ ¬Op1 ∧ ¬Op0;

  Branch = ¬S2 ∧ S1 ∧ ¬S0 ∧ ¬Op5 ∧ ¬Op4 ∧ ¬Op3 ∧ ¬Op2 ∧ Op1 ∧ ¬Op0;

  MemWrite = ¬S2 ∧ ¬S1 ∧ S0 ∧ ¬Op5 ∧ ¬Op4 ∧ Op3 ∧ ¬Op2 ∧ ¬Op1 ∧ ¬Op0;

  MemRead = (¬S2 ∧ ¬S1 ∧ ¬S0) v (¬S2 ∧ ¬S1 ∧ S0 ∧ ¬Op5 ∧ ¬Op4 ∧ ¬Op3 ∧ Op2 ∧ Op1 ∧ Op0);

  RegWrite = S2 ∧ ¬S1 ∧ ¬S0 ∧ (¬Op5 ∧ ¬Op4 ∧ ¬Op3 ∧ ¬Op2 ∧ ¬Op1 ∧ ¬Op0 ∨ ¬Op5 ∧ ¬Op4 ∧ ¬Op3 ∧ Op2 ∧ Op1 ∧ Op0 ∨ ¬Op5 ∧ ¬Op4 ∧ Op3 ∧ ¬Op2 ∧ ¬Op1 ∧ ¬Op0);

  MemtoReg = ¬S2 ∧ ¬S1 ∧ S0 ∧ ¬Op5 ∧ ¬Op4 ∧ ¬Op3 ∧ Op2 ∧ Op1 ∧ Op0;

  ALUSrc = ¬S2 ∧ S1 ∧ ¬S0 ∧ (¬Op5 ∧ ¬Op4 ∧ ¬Op3 ∧ Op2 ∧ Op1 ∧ Op0 ∨ ¬Op5 ∧ ¬Op4 ∧ Op3 ∧ ¬Op2 ∧ ¬Op1 ∧ ¬Op0);

  RegDst = ¬S2 ∧ S1 ∧ ¬S0 ∧ ¬Op5 ∧ ¬Op4 ∧ ¬Op3 ∧ ¬Op2 ∧ ¬Op1 ∧ ¬Op0.

- **(D)** F = (A ∧ B ∧ C ∧ D) ∨ (A ∧ B ∧ ¬C ∧ D) ∨ (A ∧ B ∧ ¬C ∧ ¬D) ∨ (A ∧ ¬B ∧ C ∧ D) ∨ (¬A ∧ B ∧ C ∧ D) ∨ (¬A ∧ B ∧ C ∧ ¬D) ∨ (¬A ∧ B ∧ ¬C ∧ D) ∨ (¬A ∧ ¬B ∧ ¬C ∧ D);

- **(E)** Cinco tipos de saídas serão definidas, subtração, XOR, NAND, NOR e shift de 2 bits à esquerda:

  XOR(n) = (A(n) ∧ ¬B(n)) ∨ (¬A(n) ∧ B(n));

  NAND(n) = ¬(A(n) ∧ B(n));

  NOR = ¬(A(n) ∨ B(n));

Para o subtrator de 8 bits, ele segue uma lógica similar do somador, ele possui duas saídas, a subtração e o Borrow Out e seu funcionamento também se dá em cascata:

$S(n) = A(n) \oplus B(n) \oplus \text{Borrow In}(n)$;

$\text{Borrow Out}(n) = (\neg A(n) \wedge B) \vee (\neg A(n) \wedge \text{Borrow In}(n)) \vee (B \wedge \text{Borrow In}(n))$;

A operação de shift à esquerda, não pode ser representada logicamente por meio de fórmulas booleanas.

- **(F)** Instrução do tipo R = $\text{RegDst} \wedge \neg\text{ALUSrc} \wedge \neg\text{MemtoReg} \wedge \text{RegWrite} \wedge \neg\text{MemRead} \wedge \neg\text{MemWrite}$

2.3 Verificação com Z3

Para a verificação usando o Z3, o código essencialmente seguirá essa estrutura:

- A definição das variáveis de entrada
- A expressão do circuito
- A geração de uma tabela verdade do circuito

```
#Circuito Combinacional A
from z3 import *
import itertools

# 1. Definir variáveis booleanas
x, y, z = Bools('x y z')
variaveis = [x, y, z]

# 2. Expressão booleana do circuito
F = Or(And(x, Not(And(y, z))), And(Or(Not(x), y), Not(Or(y,
And(Not(z), x)))))

# 3. Cabeçalho da tabela
print(f"{'x':<7} | {'y':<7} | {'z':<7} | {'F (Resultado)':<12}")
print("-" * 45)

# 4. Gerar todas as combinações possíveis (True/False)
for combinacao in itertools.product([True, False], repeat=3):
    val_x, val_y, val_z = combinacao

    s = Solver()
    s.add(F)

    s.add(x == val_x)
    s.add(y == val_y)
    s.add(z == val_z)

    check = s.check()
    status = "SAT" if check == sat else "UNSAT"

    resultado = (check == sat)

    print(f"{str(val_x):<7} | {str(val_y):<7} | {str(val_z):<7} |
```

```
{status:<10} | {resultado}")
```

Para o primeiro circuito analisado, começamos por definir as variáveis booleanas utilizadas, X, Y e Z. Note que, a expressão quando aplicada no código não deve ser representada puramente pelos símbolos lógicos, mas devem ser tratadas para que o Z3 interprete as fórmulas corretamente.

A geração de uma tabela verdade é opcional, pois a verificação de um estado apenas já é suficiente, contanto que respeite o comportamento correto do circuito, para cada resultado, deve se considerar o resultado True como satisfazível e False como insatisfazível .

```
#Circuito B - Somador de 8 Bits
from z3 import *
import itertools
import random

def full_adder(a, b, cin):
    # s = (a ^ b) ^ cin
    xor_ab = Or(And(a, Not(b)), And(Not(a), b))
    s = Or(And(xor_ab, Not(cin)), And(Not(xor_ab), cin))
    # cout = (a & b) | (cin & (a ^ b))
    cout = Or(And(a, b), And(cin, xor_ab))
    return s, cout

A = [Bool(f'a{i}') for i in range(8)]
B = [Bool(f'b{i}') for i in range(8)]
Cin = Bool('cin')

soma_bits = []
carry_atual = Cin
for i in range(8):
    s, carry_atual = full_adder(A[i], B[i], carry_atual)
    soma_bits.append(s)
Cout = carry_atual

def bits_to_str(bits):
    return "".join(['1' if b else '0' for b in reversed(bits)])

print(f"Verificando 10 exemplos aleatórios com Z3 Solver...\n")
print(f"{'A (bin)':<10} | {'B (bin)':<10} | {'Cin':<3} | {'Cout':<4} |
{'Soma (bin)':<10} | {'Status'}")
print("-" * 75)

for _ in range(10):
    va = [random.choice([True, False]) for _ in range(8)]
    vb = [random.choice([True, False]) for _ in range(8)]
    vcin = random.choice([True, False])

    s = Solver()

    s.add(Cin == vcin)
    for i in range(8):
        s.add(A[i] == va[i])
```

```
        s.add(B[i] == vb[i])

    if s.check() == sat:
        m = s.model()
        # Avaliar as expressões de saída baseadas no modelo encontrado
        res_soma = [is_true(m.evaluate(sb)) for sb in soma_bits]
        res_cout = is_true(m.evaluate(Cout))
        status = "SAT"
    else:
        status = "UNSAT"
        res_soma = [False]*8
        res_cout = False

    print(f"{bits_to_str(va):<10} | {bits_to_str(vb):<10} |
{int(vcin):<3} | {int(res_cout):<4} | {bits_to_str(res_soma)} |
{status}")

print(f"\nGerando arquivo 'tabela_verdade_8bits.txt'...")
print("Isso pode levar alguns segundos devido às 131.072
combinações...")

with open("tabela_verdade_8bits.txt", "w") as f:
    f.write("A(7-0) B(7-0) Cin | Cout Soma(7-0)\n")
    f.write("-" * 40 + "\n")


    for combo in itertools.product([False, True], repeat=17):
        va = combo[0:8]
        vb = combo[8:16]
        vcin = combo[16]

        # Lógica do somador aplicada à linha
        c = vcin
        res_soma = []
        for i in range(8):
            s = va[i] ^ vb[i] ^ c
            c = (va[i] & vb[i]) | (c & (va[i] ^ vb[i]))
            res_soma.append(s)
        res_cout = c

        # Escrever no arquivo (1 para True, 0 para False)
        line = f"{bits_to_str(va)} {bits_to_str(vb)} {int(vcin)} |
{int(res_cout)} {bits_to_str(res_soma)}\n"
        f.write(line)

print("Concluído! O arquivo 'tabela_verdade_8bits.txt' foi criado com
sucesso.")
```

Para o circuito somador, a abordagem continua similar ao primeiro circuito, com a definição das variáveis de entrada A, B e Cin, elas são definidas dentro um de um loop e armazenadas em um vetor.

A implementação do comportamento do somador, se dá dentro da função `full adder`, que vai conectar cada bit dos vetores de entrada A e B, assim como o Cin. É iniciado um loop de 8 instâncias para cada bit do somador.

Como a verificação de cada valor da tabela verdade do somador teriam 2^{17} (131.072) linhas, optamos por gerar 10 valores aleatórios utilizando a biblioteca random do python para realizar a checagem de satisfatibilidade do circuito no terminal do compilador. Mas o programa também gera um arquivo txt com todas as 131.072 combinações possíveis.

```
#Circuito C - Unidade de Controle MIPS multiciclo
from z3 import *
import itertools

s2, s1, s0 = Bools('s2 s1 s0')
op5, op4, op3, op2, op1, op0 = Bools('op5 op4 op3 op2 op1 op0')

# Jump: Estado 2 (010) e OpCode 000010
jump_expr = And(Not(s2), s1, Not(s0), Not(op5), Not(op4), Not(op3),
Not(op2), op1, Not(op0))

# ALUOp0 e Branch: Estado 2 (010) e OpCode 000100 (BEQ)
aluop0_expr = And(Not(s2), s1, Not(s0), Not(op5), Not(op4), Not(op3),
op2, Not(op1), Not(op0))

branch_expr = And(Not(s2), s1, Not(s0), Not(op5), Not(op4), Not(op3),
op2, Not(op1), Not(op0))

# ALUOp1 e RegDst: Estado 2 (010) e OpCode 000000 (R-Type)
aluop1_expr = And(Not(s2), s1, Not(s0), Not(op5), Not(op4), Not(op3),
Not(op2), Not(op1), Not(op0))

regdst_expr = And(Not(s2), s1, Not(s0), Not(op5), Not(op4), Not(op3),
Not(op2), Not(op1), Not(op0))

# MemWrite: Estado 1 (001) e OpCode 101011 (SW)
memwrite_expr = And(Not(s2), Not(s1), s0, op5, Not(op4), op3,
Not(op2), op1, op0)

# MemRead: Estado 0 (000) OU [Estado 1 (001) e OpCode 100011 (LW)]
memread_expr = Or(And(Not(s2), Not(s1), Not(s0)), And(Not(s2),
Not(s1), s0, op5, Not(op4), Not(op3), Not(op2), op1, op0))

# MemtoReg: Estado 1 (001) e OpCode 100011 (LW)
memtoreg_expr = And(Not(s2), Not(s1), s0, op5, Not(op4), Not(op3),
Not(op2), op1, op0)

# RegWrite: Estado 4 (100) para R-Type, LW e SW (conforme a lógica da
imagem)
regwrite_expr = And(s2, Not(s1), Not(s0), Or(And(Not(op5), Not(op4),
Not(op3), Not(op2), Not(op1), Not(op0)), And(Not(op5), Not(op4),
Not(op3), op2, op1, op0), And(Not(op5), Not(op4), op3, Not(op2),
Not(op1), Not(op0))))

# ALUSrc: Estado 2 (010) e (LW OU SW)
alusrc_expr = And(Not(s2), s1, Not(s0), Or(And(op5, Not(op4),
Not(op3), Not(op2), op1, op0), And(op5, Not(op4), op3, Not(op2), op1,
```

```
op0)))

sinais = {
    "Jump": jump_expr,
    "ALUOp0": aluop0_expr,
    "Branch": branch_expr,
    "MemWrite": memwrite_expr,
    "MemRead": memread_expr,
    "RegWrite": regwrite_expr,
    "ALUSrc": alusrc_expr
}

opcodes_teste = {
    "R-Type (000000)": (False, False, False, False, False, False),
    "Jump   (000010)": (False, False, False, False, True, False),
    "BEQ    (000100)": (False, False, False, True, False, False),
    "LW     (100011)": (True, False, False, False, True, True),
    "SW     (101011)": (True, False, True, False, True, True)
}


print(f"{'Estado':<8} | {'Instrução (OpCode)':<20} | {'Sinal
Ativo':<12} | {'Z3 Status'}")
print("-" * 65)


# Testando nos estados principais S0, S1, S2 e S4
for v_s2, v_s1, v_s0 in [(False,False,False), (False,False,True),
(False,True,False), (True,False,False)]:
    for nome_inst, v_ops in opcodes_teste.items():
        v_op5, v_op4, v_op3, v_op2, v_op1, v_op0 = v_ops

        for nome_sinal, expr in sinais.items():

            s = Solver()
            s.add(expr)
            # Adiciona as restrições do cenário atual
            s.add(s2 == v_s2, s1 == v_s1, s0 == v_s0)
            s.add(op5 == v_op5, op4 == v_op4, op3 == v_op3, op2 ==
v_op2, op1 == v_op1, op0 == v_op0)


            if s.check() == sat:
                est_str = f"S{int(v_s2)*4 + int(v_s1)*2 + int(v_s0)}"
                print(f"{est_str:<8} | {nome_inst:<20} |
{nome_sinal:<12} | SAT (Ativo)")
```

Os sinais de entrada aqui, são definidos pelos Opcodes das operações que entram na unidade de controle (Inst 31:26), representados por Op0 até Op5, e os estados da máquina FSM, estes que são adaptações dos sinais de controle de 1 e de 2 bits, referenciados por Patterson & Hennessy no capítulo 5 de seu livro “Organização e Projeto De Computadores.”, aqui representados de S0 à S2.

O tratamento dos sinais de saída é o mesmo dos circuitos anteriores, a fórmula booleana é adaptada para a sintaxe do Z3, e então são feitas as checagens que serão as

as instruções que são executadas pela unidade de controle: Tipo-R, Jump, Back If Equal, Load Word e Store Word.

```
#Circuito D - Circuito Combinacional
from z3 import *
import itertools

a, b, c, d = Bools('a b c d')
variaveis = [a, b, c, d]

# Expressão booleana do circuito.
F = 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))

print(f"{'a':<7} | {'b':<7} | {'c':<7} | {'d':<7} |{'F
(Resultado)':<12}")
print("-" * 55)

# Gerar todas as combinações possíveis (True/False).
for combinacao in itertools.product([False, True], repeat=4):
    val_a, val_b, val_c, val_d = combinacao

    resultado = simplify(substitute(F, (a, BoolVal(val_a)), (b,
BoolVal(val_b)), (c, BoolVal(val_c)),(d, BoolVal(val_d))))

    # Imprimir a linha da tabela
    print(f"{str(val_a):<7} | {str(val_b):<7} | {str(val_c):<7} |
{str(val_d):<7} | {str(resultado):<12}")
```

Muito similar à abordagem do primeiro circuito combinacional visto, não há muitas diferenças quanto a eles, o Circuito D utiliza o mesmo código e o mesmo processo do Circuito A.

```
#Circuito E - ULA de 8 bits
from z3 import *
import itertools

a, b, b_in = Bools('a b b_in')

def imprimir_separador(titulo):
    print(f"\n{'='*20} {titulo} {'='*20}")

def resolver_linha(expressao, vars_map):
    """Auxiliar para verificar se a combinação é SAT ou UNSAT na
expressão"""
    s = Solver()
    s.add(expressao)
    for var, val in vars_map.items():
        s.add(var == val)

    check = s.check()
    return "SAT" if check == sat else "UNSAT", (check == sat)
```

```
imprimir_separador("TABELA VERDADE: SUBTRAÇÃO (1 BIT)")

# Definições das expressões.
Diferenca = Xor(Xor(a, b), b_in)
Emprestimo = Or(And(Not(a), b), And(Not(Xor(a, b)), b_in))

print(f"{'A':<6} | {'B':<6} | {'Bin':<6} | {'Dif (Status)':<12} |
{'Empr (Status)':<12}")
print("-" * 65)

for comb in itertools.product([True, False], repeat=3):
    v_a, v_b, v_bin = comb
    mapeamento = {a: v_a, b: v_b, b_in: v_bin}

    st_d, res_d = resolver_linha(Diferenca, mapeamento)
    st_e, res_e = resolver_linha(Emprestimo, mapeamento)

    print(f"{str(v_a):<6} | {str(v_b):<6} | {str(v_bin):<6} |
{st_d:<12} | {st_e:<12}")


imprimir_separador("TABELA VERDADE: FUNÇÕES LÓGICAS")
f_xor  = Xor(a, b)
f_nand = Not(And(a, b))
f_nor  = Not(Or(a, b))

print(f"{'A':<6} | {'B':<6} | {'XOR (SAT)':<10} | {'NAND (SAT)':<10} |
{'NOR (SAT)':<10}")
print("-" * 55)

for comb in itertools.product([True, False], repeat=2):
    v_a, v_b = comb
    mapeamento = {a: v_a, b: v_b}


    st_xor, _  = resolver_linha(f_xor, mapeamento)
    st_nand, _ = resolver_linha(f_nand, mapeamento)
    st_nor, _  = resolver_linha(f_nor, mapeamento)

    print(f"{str(v_a):<6} | {str(v_b):<6} | {st_xor:<10} |
{st_nand:<10} | {st_nor:<10}")


imprimir_separador("MAPEAMENTO DE HARDWARE: SHIFT LEFT 2")
print(f"{'Posição Saída':<15} | {'Origem do Dado':<20}")
print("-" * 40)

for i in range(8):
    origem = "Fixo em 0 (GND)" if i < 2 else f"Vem do Bit {i-2} de
entrada"
    print(f"Bit {i:<11} | {origem}")
```

A abordagem desse circuito é muito similar ao Somador de 8 bits, mas por escalabilidade, foram reduzidas para operações básicas com apenas 1 bit.

A função de subtração foi definida manualmente, onde as saídas de Borrow e subtração seguem uma implementação similar à do somador de 8 bits, onde a função

XOR foi definida com os operadores And, Or e Not. Quanto às operações XOR, NAND, e NOR seguem uma implementação equivalente utilizando os mesmos operadores do subtrator; Quanto a operação de shift de 2 bits, ela não pode ser representada de uma maneira booleana, então a solução encontrada foi definir quais bits individuais ocupação suas novas posições após a operação.

```
#Circuito F - Instrução do tipo R
from z3 import *

RegDst = Bool('RegDst')
ALUSrc = Bool('ALUSrc')
MemtoReg = Bool('MemtoReg')
RegWrite = Bool('RegWrite')
MemRead = Bool('MemRead')
MemWrite = Bool('MemWrite')

# F representa a ativação correta do caminho de dados para Tipo R
fluxo_tipo_R = And(RegDst, Not(ALUSrc), Not(MemtoReg), RegWrite,
Not(MemRead), Not(MemWrite))

solver = Solver()
solver.add(fluxo_tipo_R)

if solver.check() == sat:
    print("O fluxo da instrução Tipo R é logicamente possível.")
    print("Exemplo de estado:", solver.model())
else:
    print("O fluxo configurado é impossível (conflito de sinais).")
```

Aqui os sinais de entrada são representados pelos sinais de controle que saem da Unidade de Controle e são ativados durante o fluxo de instrução. Para cada um foi definido se o sinal está ativo (True) ou não (False).

3. Processo de simplificação e verificação de redundância

Após a verificação de que cada modelagem dos circuitos estava coerente, a maneira como cada um foi modelado inicialmente passou por um processo de verificação de redundância, utilizando um programa em python utilizando a biblioteca Sympy, que tem o objetivo de reduzir as expressões lógicas utilizadas para as tornar mais eficiente, sem alterar o seu resultado.

```
#Circuito A
from sympy import sympify, simplify_logic

def simplificar_string_booleana(formula_str):
    expressao = sympify(formula_str)

    expressao_simplificada = simplify_logic(expressao)

    return str(expressao_simplificada)
```

```
entrada = "Or(And(x, Not(And(y, z))), And(Or(Not(x), y), Not(Or(y,
And(Not(z), x)))))"

saida = simplificar_string_booleana(entrada)

saida_bool = saida.replace("|", " ∨").replace("&", "∧").replace("~",
"¬")

print(f"Fórmula original: {entrada}")
print(f"Fórmula simplificada:   {saida_bool}")
```

O processo para todos os testes é similar, primeiro definimos uma função de simplificação do circuito usando as ferramentas do Sympy, a expressão utilizada no circuito original é alimentada. Como a função sympify devolve a fórmula reduzida em outros símbolos, eles são ajustados em seguida para ficar no padrão booleano.

O resultado de cada simplificação foi:

Tabela 1. Tabela dos resultados das simplificações utilizando a ferramenta Sympy.

Fórmula Original		**Fórmula Simplificada**
(A)	F = ((¬X ∧ Y) ∧ ¬(Y ∨ (X ∧ ¬Z))) ∨ (X ∨ ¬(Y ∧ Z))	¬Y ∨ (X ∧ ¬Z)
(B)	S = A ⊕ B ⊕ Carry In Carry Out = (A ∧ B) ∨ (Carry In ∧ (A ∨ B))	O circuito já se encontra na sua forma mais otimizada. (Fórmula igual ou resultado maior que o original)
(C)	Jump = ¬S2 ∧ S1 ∧ ¬S0 ∧ ¬Op5 ∧ ¬Op4 ∧ ¬Op3 ∧ ¬Op2 ∧ Op1 ∧ Op0; ALUOp0 = ¬S2 ∧ S1 ∧ ¬S0 ∧ ¬Op5 ∧ ¬Op4 ∧ ¬Op3 ∧ ¬Op2 ∧ Op1 ∧ ¬Op0; ALUOp1 = ¬S2 ∧ S1 ∧ ¬S0 ∧ ¬Op5 ∧ ¬Op4 ∧ ¬Op3 ∧ ¬Op2 ∧ ¬Op1 ∧ ¬Op0; Branch = ¬S2 ∧ S1 ∧ ¬S0 ∧ ¬Op5 ∧ ¬Op4 ∧ ¬Op3 ∧ ¬Op2 ∧ Op1 ∧ ¬Op0;	O circuito já se encontra na sua forma mais otimizada. (Fórmula igual ou resultado maior que o original)

	MemWrite = ¬S2 ∧ ¬S1 ∧ S0 ∧ ¬Op5 ∧ ¬Op4 ∧ Op3 ∧ ¬Op2 ∧ ¬Op1 ∧ ¬Op0; MemRead = (¬S2 ∧ ¬S1 ∧ ¬S0) v (¬S2 ∧ ¬S1 ∧ S0 ∧ ¬Op5 ∧ ¬Op4 ∧ ¬Op3 ∧ Op2 ∧ Op1 ∧ Op0); RegWrite = S2 ∧ ¬S1 ∧ ¬S0 ∧ (¬Op5 ∧ ¬Op4 ∧ ¬Op3 ∧ ¬Op2 ∧ ¬Op1 ∧ ¬Op0 ∨ ¬Op5 ∧ ¬Op4 ∧ ¬Op3 ∧ Op2 ∧ Op1 ∧ Op0 ∨ ¬Op5 ∧ ¬Op4 ∧ Op3 ∧ ¬Op2 ∧ ¬Op1 ∧ ¬Op0); MemtoReg = ¬S2 ∧ ¬S1 ∧ S0 ∧ ¬Op5 ∧ ¬Op4 ∧ ¬Op3 ∧ Op2 ∧ Op1 ∧ Op0; ALUSrc = ¬S2 ∧ S1 ∧ ¬S0 ∧ (¬Op5 ∧ ¬Op4 ∧ ¬Op3 ∧ Op2 ∧ Op1 ∧ Op0 ∨ ¬Op5 ∧ ¬Op4 ∧ Op3 ∧ ¬Op2 ∧ ¬Op1 ∧ ¬Op0); RegDst = ¬S2 ∧ S1 ∧ ¬S0 ∧ ¬Op5 ∧ ¬Op4 ∧ ¬Op3 ∧ ¬Op2 ∧ ¬Op1 ∧ ¬Op0.	
(D)	F = (A ∧ B ∧ C ∧ D) ∨ (A ∧ B ∧ ¬C ∧ D) ∨ (A ∧ B ∧ ¬C ∧ ¬D) ∨ (A ∧ ¬B ∧ C ∧ D) ∨ (¬A ∧ B ∧ C ∧ D) ∨ (¬A ∧ B ∧ C ∧ ¬D) ∨ (¬A ∧ B ∧ ¬C ∧ D) ∨ (¬A ∧ ¬B ∧ ¬C ∧ D);	F = (A ∧ C ∧ D) ∨ (A ∧ B ∧ ¬C) ∨ (B ∧ C ∧ ¬A) ∨ (D ∧ ¬A ∧ ¬C)
(E)	S(n) = A(n) ⊕ B(n) ⊕ Borrow In(n) ; Borrow Out(n) = (¬A(n) ∧ B) ∨ (¬A(n) ∧ Borrow In(n)) ∨ (B ∧ Borrow In(n)); XOR(n) = (A(n) ∧ ¬B(n)) ∨ (¬A(n) ∧ B(n));	O circuito já se encontra na sua forma mais otimizada. (Fórmula igual ou resultado maior que o original)

	NAND(n) = ¬(A(n) ∧ B(n)); NOR = ¬(A(n) ∨ B(n));	
(F)	Instrução do tipo R = RegDst ∧ ¬ALUSrc ∧ ¬MemtoReg ∧ RegWrite ∧ ¬MemRead ∧ ¬MemWrite.	O circuito já se encontra na sua forma mais otimizada. (Fórmula igual ou resultado maior que o original)

Os casos em que a fórmula após a simplificação são maiores que a original, ocorrem quando portas mais complexas vão para formas reduzidas, como por exemplo A XOR B, pode ser "reduzido" para (A ∧ ¬B) ∨ (¬A ∧ B);

3.1 Verificação das simplificações

Após a verificação à procura de redundâncias, os resultados passaram por um processo de verificação igual aos originais, para checar se não houve nenhuma alteração nos resultados. A atenção deve ser voltada principalmente aos circuitos A e D, pois foram os únicos que sofreram maiores alterações:

```
#Circuito A simplificado
from z3 import *
import itertools

x, y, z = Bools('x y z')
F = Or(Not(y), And(x, Not(z)))

print(f"{'x':<7} | {'y':<7} | {'z':<7} | {'Status':<10} |
{'Resultado'}")
print("-" * 55)

for combinacao in itertools.product([True, False], repeat=3):
    val_x, val_y, val_z = combinacao

    s = Solver()

    s.add(F)

    s.add(x == val_x)
    s.add(y == val_y)
    s.add(z == val_z)

    check = s.check()
    status = "SAT" if check == sat else "UNSAT"

    resultado = (check == sat)

    print(f"{str(val_x):<7} | {str(val_y):<7} | {str(val_z):<7} |
```

```
{status:<10} | {resultado}")
```

Como exemplo, o código do circuito A simplificado. Note que a estrutura é igual ao código utilizando a fórmula original para manter um padrão de resultados.

3.2 Tabela da Verificação dos Circuito

Tabela 2. Tabela do Circuito A.

x	y	z	F (Status)	Resultado
True	True	True	UNSAT	False
True	True	False	SAT	True
True	False	True	SAT	True
True	False	False	SAT	True
False	True	True	UNSAT	False
False	True	False	UNSAT	False
False	False	True	SAT	True
False	False	False	SAT	True

Tabela 3. Tabela do Circuito B (valores gerados aleatoriamente).

A (bin)	B (bin)	Cin	Cout	Soma (bin)	Status
00101111	00011001	1	0	01001001	SAT
10100010	00000000	0	0	10100010	SAT
01001011	11000010	0	1	00010101	SAT
10100010	00111101	0	0	11011111	SAT
11010000	01001010	0	1	00011010	SAT
10101011	01011111	1	1	00001010	SAT
01110000	10000010	0	0	11110010	SAT
00000000	10000010	0	0	10000010	SAT
01110101	01111011	0	0	11110000	SAT
10000110	10111111	0	1	01000101	SAT

Tabela 4. Tabela do Circuito C.

Estado	Instrução	Sinal Ativo	Status
S0	R-Type (000000)	MemRead	SAT
S0	Jump (000010)	MemRead	SAT
S0	BEQ (000100)	MemRead	SAT
S0	LW (100011)	MemRead	SAT
S0	SW (101011)	MemRead	SAT
S1	LW (100011)	MemRead	SAT
S1	SW (101011)	MemWrite	SAT
S2	Jump (000010)	Jump	SAT
S2	BEQ (000100)	ALUOp0	SAT
S2	BEQ (000100)	Branch	SAT
S2	LW (100011)	ALUSrc	SAT
S2	SW (101011)	ALUSrc	SAT
S4	R-Type (000000)	RegWrite	SAT

Tabela 5. Tabela do Circuito D.

a	b	c	d	F
False	False	False	False	False
False	False	False	True	True
False	False	True	False	False
False	False	True	True	False
False	True	False	False	False
False	True	False	True	True
False	True	True	False	True

False	True	True	True	True
True	False	False	False	False
True	False	False	True	False
True	False	True	False	False
True	False	True	True	True
True	True	False	False	True
True	True	False	True	True
True	True	True	False	False
True	True	True	True	True

Tabela 6. Tabela Verdade: Subtração (1 Bit), do Circuito E.

A	B	Bin	Dif (Status)	Empr (Status)
True	True	True	SAT	SAT
True	True	False	UNSAT	UNSAT
True	False	True	UNSAT	UNSAT
True	False	False	SAT	UNSAT
False	True	True	UNSAT	SAT
False	True	False	SAT	SAT
False	False	True	SAT	SAT
False	False	False	UNSAT	UNSAT

Tabela 7. Tabela Verdade: Funções Lógicas, do Circuito E.

A	B	XOR (SAT)	NAND (SAT)	NOR (SAT)

True	True	UNSAT	UNSAT	UNSAT
True	False	SAT	SAT	UNSAT
False	True	SAT	SAT	UNSAT
False	False	UNSAT	SAT	SAT

Tabela 8. Mapeamento de Hardware: Shift Left 2, do Circuito E.

Posição Saída	Origem do Dado
Bit 0	Fixo em 0 (GND)
Bit 1	Fixo em 0 (GND)
Bit 2	Vem do Bit 0 de entrada
Bit 3	Vem do Bit 1 de entrada
Bit 4	Vem do Bit 2 de entrada
Bit 5	Vem do Bit 3 de entrada
Bit 6	Vem do Bit 4 de entrada
Bit 7	Vem do Bit 5 de entrada

Tabela 9. Tabela do Circuito F, Exemplo de Estado.

Sinal	Estado
MemRead	False
RegDst	True

MemtoReg	False
MemWrite	False
ALUSrc	False
RegWrite	True

3.3 Comparação entre fórmulas

Para o processo final de verificação entre os circuitos originais e os reduzidos, é feito um último teste utilizando as fórmulas. Novamente, com o Z3, vamos verificar se as fórmulas são equivalentes logicamente utilizando o solver. Para todos os casos, se houver alguma diferença de satisfatibilidade, os circuitos não são equivalentes. A estrutura geral de cada comparação segue este padrão:

```
#Comparador dos Circuitos A
from z3 import *

x, y, z = Bools('x y z')

F_original = Or(And(x, Not(And(y, z))), And(Or(Not(x), y), Not(Or(y,
And(Not(z), x)))))
F_simplificado = Or(Not(y), And(x, Not(z)))


solver = Solver()

solver.add(F_original != F_simplificado)

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.")
```

As variáveis booleanas são criadas no início e em seguida, a fórmula original e simplificada são definidas. O solver do Z3 vai verificar se alguma saída delas é diferente e então a execução é finalizada. Após todas as verificações esses são os resultados:

Tabela 10. Tabela de comparações entre os circuitos.

Circuito original	Circuito Simplificado	Status
(A)	(A)	Satisfatível

(B)	(B)	Sem Alterações (Satisfatível)
(C)	(C)	Sem Alterações (Satisfatível)
(D)	(D)	Satisfatível
(E)	(E)	Sem Alterações (Satisfatível)
(F)	(F)	Sem Alterações (Satisfatível)

4. Conclusão

Este trabalho apresentou uma metodologia sistemática para a verificação formal de circuitos digitais por meio da modelagem em lógica booleana e do uso combinado das bibliotecas Z3 e SymPy. A abordagem demonstrou ser eficaz tanto na validação funcional dos circuitos quanto na identificação de redundâncias e oportunidades de simplificação lógica, mantendo rigorosamente a equivalência comportamental entre as versões original e simplificada.

A análise de seis circuitos distintos, abrangendo desde circuitos combinacionais simples até componentes mais complexos como uma Unidade Lógica e Aritmética, uma Unidade de Controle baseada em Máquina de Estados Finitos e o fluxo de execução de uma instrução do tipo R no MIPS, evidenciou que a modelagem formal permite verificar automaticamente propriedades que seriam custosas ou propensas a erro se realizadas manualmente. Em especial, o uso do Z3 possibilitou a prova automática de satisfatibilidade e equivalência lógica, enquanto o SymPy se mostrou adequado para a manipulação simbólica e a redução de expressões booleanas.

Os resultados indicaram que, em diversos casos, os circuitos já se encontravam em formas logicamente otimizadas, e que tentativas de simplificação nem sempre produzem expressões menores, especialmente quando operadores de alto nível, como XOR, são expandidos em combinações de portas lógicas básicas. Ainda assim, o processo de simplificação seguido de verificação formal garantiu que qualquer transformação aplicada não alterasse o comportamento funcional dos circuitos, o que é fundamental em contextos de projeto e otimização de hardware.

Dessa forma, conclui-se que a verificação formal baseada em lógica booleana e solucionadores SMT é uma ferramenta poderosa para aumentar a confiabilidade do projeto de circuitos digitais, reduzir erros de especificação e apoiar processos de otimização segura.

References

GHOSH, Sukumar. Computer Architecture: Basics. Lecture Notes (CS:60:16). Iowa City: University of Iowa, 2010. Disponível em: https://homepage.divms.uiowa.edu/~ghosh/6016.90.pdf. Acesso em: 5 jan. 2026.

MICROSOFT RESEARCH. Z3 Theorem Prover. 2026. Versão 4.13.0. Disponível em: https://github.com/Z3Prover/z3. Acesso em: 20 dez. 2025.

PANNAIN, Ricardo de Oliveira. Arquitetura de Computadores: Hierarquia de Memória. Notas de aula da disciplina MC542 (Organização de Computadores e Linguagem de Montagem). Campinas: IC-UNICAMP, [20--]. Disponível em: https://www.ic.unicamp.br/~pannain/mc542/aulas/ch5_arq.pdf. Acesso em: 4 jan. 2026.

PATTERSON, David A.; HENNESSY, John L. Organização e projeto de computadores. Tradução de Daniel Vieira. 4. ed. Rio de Janeiro: Elsevier, 2014.

PYTHON SOFTWARE FOUNDATION. itertools — Functions creating iterators for efficient looping. 2026. Python 3.12 documentation. Disponível em: https://docs.python.org/3/library/itertools.html. Acesso em: 20 dez. 2025.

SYMPY DEVELOPMENT TEAM. SymPy: Symbolic computing in Python. 2026. Disponível em: https://github.com/sympy/sympy. Acesso em: 20 dez. 2025.

TORONTO METROPOLITAN UNIVERSITY. Multicycle Control. Course Lectures (COE608: Computer Organization and Architecture). Toronto: Department of Electrical, Computer, and Biomedical Engineering, [20--]. Disponível em: https://www.ee.torontomu.ca/~courses/coe608/lectures/Multicycle-Control.pdf. Acesso em: 8 jan. 2026.