Metastability-Containing Hardware Final Project

Yaniv Hajaj

Asaf Ben Or

https://github.com/YanivHajaj/Metastability-Containing-Hardware-Final-Project/tree/main

תיאור הפרויקט:

הקוד מיישם מערכת לזיהוי מטא-סטביליות בביטסטים (bitsets) ובוחן ביטויים לוגיים המבוססים על ביטסטים אלו. המערכת יכולה להתמודד עם ביטסטים המכילים ביטים מטא-סטביליים M ויכולה להעריך את ערך הביטויים בהתאם לתוצאות המטא-סטביליות. הקוד גם כולל פונקציות להרחבת ביטסטים, יצירת קומבינציות, והערכת ביצועים של המעגלים הלוגיים החסינים למטאסטיבליות.

התכנית תקבל כקלט פונקציה בוליאנית (n ביטים) אשר הפלט שלה יהיה ביט בודד, ובנוסף תקבל כקלט את K שהוא מספר הביטים שבהם יכול להיות M (לא יודעים באיזה ביטים ספציפיים).

הקלט יהיה: $\{0,1,M\}^n$ והפלט יהיה: $F_m(x) =* F(res(x))$

זוהי כמובן הפונקציה האידיאלית ולא ניתן להשיג יותר טוב ממנה.

כעת נשתמש בבנייה:

Two staged Construction [Ikenmeyer et al. JACM'19] מההרצאה.

First stage (similar to the general exp. size construction)

Second stage (going over all k-subsets of indices in a smart way)

Two staged Construction [Ikenmeyer et al. JACM'19]

- First stage (similar to the general exp. size construction):
  - Assume that the k M-bits appear at fixed positions (e.g., at indices1,4,9,20,200…).
  - Take 2^k copies of the circuit C.
  - For the ith copy, fix the k fixed bits to the binary representation of i.
  - Now, use a CMUX to select one of these 2^k outputs, where the original k input bits that we replaced are used as the select bits.
  - The following lemma is proven similarly to the exp. Size general case. The size bound is trivial (i.e., 2^k copies of C).

Lemma 6.8. *Let C be a circuit implementing $f : \{0,1\}^n \to \{0,1\}$ and $S \subseteq [n]$ with $|S| = k$. Denote by $|C|$ the size (i.e., number of gates) of C. Then there is a circuit of size at most $2^k(|C| + \mathcal{O}(1))$ that computes $f_{\mathrm{M}}(x)$ for any $x \in \{0,1,\mathrm{M}\}$ satisfying that $x_i = \mathrm{M} \Rightarrow i \in S$.*

Two staged Construction, cont.

- Second stage (going over all k-subsets of indices in a smart way):
  - Order all k-subsets of the indices set $\{1, \dots, n\}$
  - The number of sets in that ordering is $I \triangleq \binom{n}{k}$
  - Let S_i denote the ith set, where $i \in \{1, \dots, I\}$
    - Example: For the indices set $\{1,2,3\}$ we (can) have the following enumeration of 2-subsets: $S_1: \{1,2\}, S_2: \{1,3\}, S_3: \{2,3\}$
  - Let C_{ij} denote the circuit from the 1st stage where the M bits are in $S_i \cup S_j$
    - That is, one should plug in (at most) $2k$ in the 1st stage's construction.
  - Let $a_i \triangleq \mathrm{AND}_{\mathrm{M}}(C_{i1}, \dots, C_{iI})$ for $i \in \{1, \dots, I\}$.
  - The output of our circuit is $o \triangleq \mathrm{OR}_{\mathrm{M}}(a_1, \dots, a_I)$
    - Both $\mathrm{AND}_{\mathrm{M}}$ and OR_{M} over I inputs are implemented by a tree of size $2I - 1$ and depth $O(\log I)$.
- Total depth of the construction:
  - $depth(trees) + depth(2^{2k}: 1 - \mathrm{CMUX}) + depth(C) = O(\log I) + O(k) + depth(C) =$ $O(k \log \frac{n}{k} + depth(C))$

$\binom{n}{k} \le \left(\frac{en}{k}\right)^k$

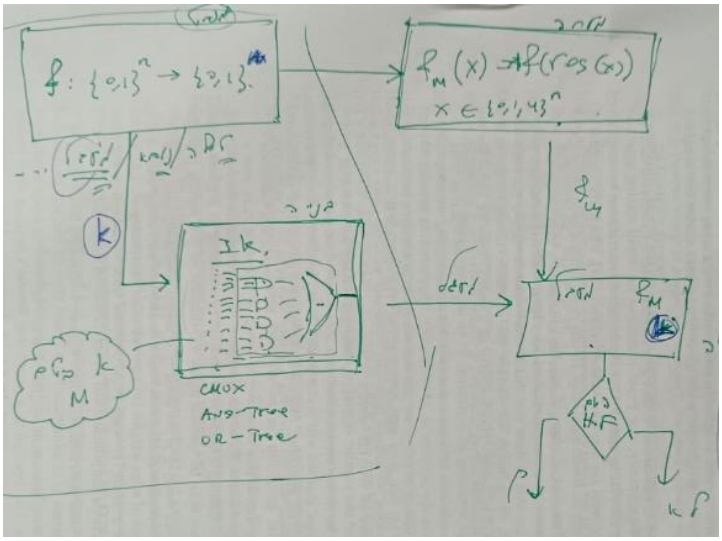
31

פרטים טכניים:

המעגל יבנה ע"י Cmux AND Tree וTree or כפי שראינו בהרצאות.
נריץ את כל אופציות הימצאות K הביטים הבעיתיים ואת כלל אפשרויות המעגל ונקבל ביט פלט שהוא 0 1 או M .
נבדוק כמות ביטים של M שהיא קטנה או שווה לK ואם אנו מקבלים את אותה תוצאה כמו בחלק 1.
יוחזרו נתונים כמו עומק המעגל שבנינו ן מספר השערים, יוחזרו נתונים לבדיקה של פעולת המעגל במקרה שבו K גדול מהK שעל בסיסו נבנה המעגל.

הקלט תהיה פונקצייה בוליאנית בצורה של נוסחא כגון: a*b+c שזה $(a\ And\ b) OR\ c$ והפלט הוא ביט בודד.

באיור למטה ניתן לראות את המסלול הימני שם נחשב את F_M שזו הפונקציה המדויקת, ובצד שמאל שהו המסלול של המעגל שבנינו, לבסוף אנחנו משווים בינהם ובודקים איך הוא מתפקד כתלות בכמה ביטים מטאסטיבליים נכנסים אליו (מיוצג על ידי K)



נציג את הבנייה שלנו ואת הפונקציות השונות:

בשלב ראשון נבנה פונקציה שמקבלת bitset על ידי מערך ביטים והיא מזהה מטאטסטיביליות :
detect_metastability

מקבלת רשימה של ביטסטים ובודקת בכל עמדת ביט האם כל הביטים זהים או לא. אם הביטים אינם זהים, הפונקציה מסמנת את העמדה כמטא-סטבילית ('M'), אחרת היא משתמשת בערך הביט השווה.

```python
def detect_metastability(bitsets):
    # Determine the length of the bitsets
    bit_length = len(bitsets[0])

    # Initialize the result with zeros
    result = ['0'] * bit_length

    # Check each bit position for metastability
    for i in range(bit_length):
        # Extract the ith bit from each bitset
        bits_at_i = [bitset[i] for bitset in bitsets]

        # If not all bits are the same, mark as metastable
        if len(set(bits_at_i)) > 1:
            result[i] = 'M'
        else:
            # Otherwise, use the bit value (all the same)
            result[i] = bits_at_i[0]

    return ''.join(result)

# # Given bitsets
# bitsets = ["1","1","1","1"]
```

לדוגמה אם הביטים הם ["1", "1", "1"], כולם זהים. לכן, המיקום 0 יציב עם ערך הביט '1'.
אם הביטים הם ["1", "1", "0"], לא כולם זהים. לכן, המיקום 1 הוא מטסטבילי, ואנו מסמנים אותו בM.

פונקציה נוספת לוקחת את כל הקומבינציות שאפשר מביטוי כמו MM0 מחזירה את קבוצת הרזולוציה:

generate_combinations_list מייצרת את כל הקומבינציות האפשריות עבור ביטסט המכיל ביטים מטא-סטביליים ('M'). הפונקציה מחליפה את כל 'M' בשני האפשרויות ('0' ו-'1') ויוצרת רשימה של כל הקומבינציות האפשריות.

```python
def generate_combinations_list(bitset):
    # Replace 'M' in bitset with both possibilities '0' and '1'
    combinations = []
    if 'M' in bitset:
        # Generate combinations for 'M' by replacing it with '0' and '1'
        for bit in ['0', '1']:
            combinations += generate_combinations_list(bitset.replace('M', bit, 1))
    else:
        # No 'M' found, add the bitset as is
        combinations.append(bitset)

    return combinations

# # Example bitset with metastability
# bitset_example = "MM0"
```

פונקציה נוספת פותרת ביטוי בוליאני ללא M של מטאסטיבליות:
הפונקציה עושה שימוש בספריית sympify שיודעת להתמודד עם ביטויים בוליאנים.
plot_bitset_expression_result משתמשת בסימבולים כדי למפות את הביטסט לערכים בוליאניים ומעריכה את הביטוי הלוגי בהתאם לערכים אלו.

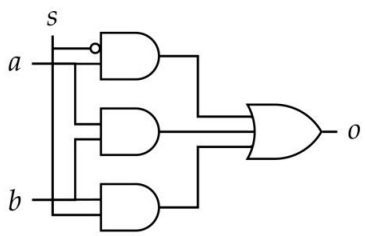
```python
def plot_bitset_expression_result(bitset, expression):
    # Define symbols for the expression (up to 26 variables)
    variables = symbols('a:z')
    # Map bitset to variable values
    values = {variables[i]: bool(int(bitset[i])) for i in range(len(bitset))}

    # Evaluate the expression
    expr = sympify(expression)
    result = expr.subs(values)

    t=0

    if (result):
        t=1

    return(t)

# # Example usage
# bitset = "10"
# expression = "~a&~b"
```

פונקציה נוספת מסכמת את חלק א, מקבלת bitset שיש בו M מטאסטיבילי וביטוי ויודעת להוציא את כל הקומבינציות מקבוצת הרזולציה, ושולחת כל אחת מהאופציות עם הביטוי הבוליאני.
את התוצאות היא לוקחת ומכניסה לפונקציה של זיהוי מטאסטביליות ובסוף נקבל 0,1,M כתוצאה שזוהי הפונקציה המדוייקת.
integrated_metastability_detection - משלבת את פונקציות יצירת הקומבינציות והערכת הביטוי כדי לזהות מטא-סטביליות עבור ביטסט נתון המכיל ביטים מטא-סטביליים ('M'). היא מייצרת את כל הקומבינציות, מעריכה את הביטוי עבור כל קומבינציה, ובודקת את המטא-סטביליות של התוצאות.

```python
def integrated_metastability_detection(bitset_with_m, expression):
    # Generate all combinations from the bitset with 'M'.
    bitsets = generate_combinations_list(bitset_with_m)

    # Evaluate each bitset against the expression and collect results.
    results = [str(plot_bitset_expression_result(bitset, expression)) for bitset in bitsets]

    # Analyze the results for metastability.
    metastability_result = detect_metastability(results)

    # Return the metastability detection result.
    return metastability_result
```

כעת בנינו CMUX 2:1 בעזרת הפונקציה הבאה:

- Second shot
  - $\mathrm{MUX}(1,1,u) = 1 \Rightarrow$ **hazard free** ✓



ab \ s	0	1	M
00	0	0	0
01	0	1	M
10	1	0	M
11	1	1	1
0M	0	M	M
M0	M	0	M
1M	1	M	M
M1	M	1	M
MM	M	M	M

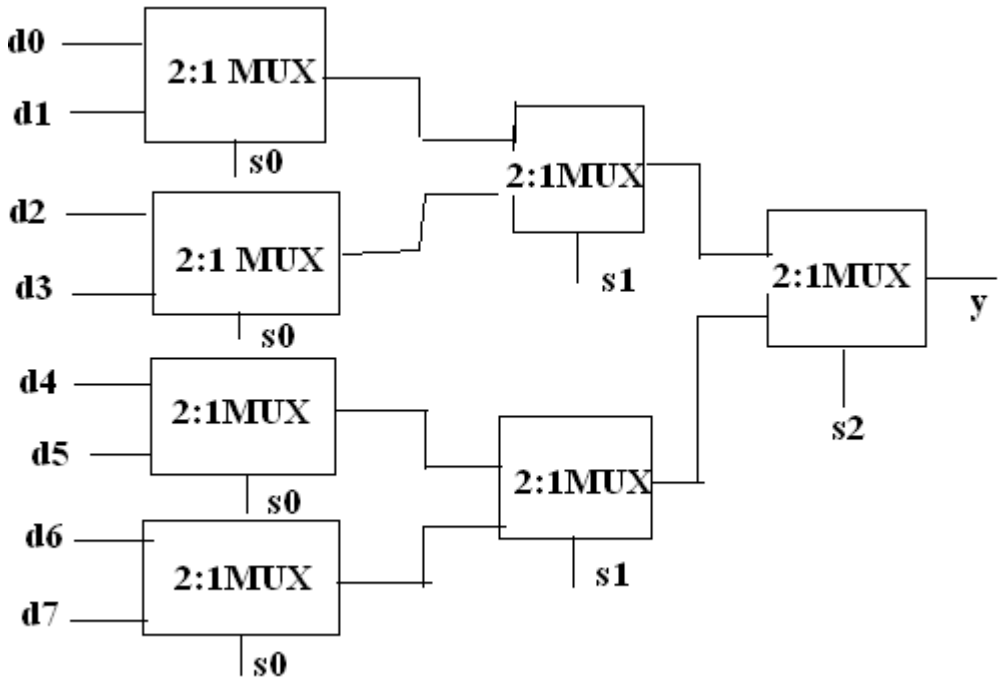
מדמה פעולת מולטיפלקסר 2-ל-1 כפי שראינו בשיעור. היא משתמשת בביטים 'a', 'b' ובחירת הביט 's' כדי לחשב את תוצאת ה-MUX בעזרת פונקציות זיהוי המטא-סטביליות.

```python
def cmux2_1(a,b,s):
    bitset=str(a)+str(b)
    used_bitset=bitset+str(s)
    results=integrated_metastability_detection(used_bitset,"a&~c|b&c|a&b")
    return results
```

```
al/Part1.py"
[('0', '0', '0', '0'), ('0', '0', '1', '0'), ('0', '0', 'M', '0'), ('0', '1', '0', '0'), ('0', '1', '1', '1'), ('0', '1', 'M', 'M'),
('0', 'M', '0', '0'), ('0', 'M', '1', 'M'), ('0', 'M', 'M', 'M'), ('1', '0', '0', '1'), ('1', '0', '1', '0'), ('1', '0', 'M', 'M'), (
'1', '1', '0', '1'), ('1', '1', '1', '1'), ('1', '1', 'M', '1'), ('1', 'M', '0', '1'), ('1', 'M', '1', 'M'), ('1', 'M', 'M', 'M'), ('
M', '0', '0', 'M'), ('M', '0', '1', '0'), ('M', '0', 'M', 'M'), ('M', '1', '0', 'M'), ('M', '1', '1', '1'), ('M', '1', 'M', 'M'), ('M
', 'M', '0', 'M'), ('M', 'M', '1', 'M'), ('M', 'M', 'M', 'M')]
```

ניסינו את כל האופציות של M,0,1 ואכן כפי שניתן לראות הבנייה הצליחה כיוון שכל אופציה מחזירה את הערך הנכון בטבלת האמת של CMUX שראינו בהרצאה (למשל כניסה של a=0 b=1 s=M אכן מחזירה M כדרוש)

כעת בנינו MUX N:1 על ידי בנייה רקורסיבית (בדומה לדוגמה הבאה של פירוק CMUX של 8:1 לכאלה של 2:1



הבנייה משתמשת בהבנה שכל פעם אנחנו מחלקים את הקלטים שלנו ב2 והופכים את הבעיה לקלה יותר. כלומר בסיבוב הראשון נפתור בעיה דומה של שני CMU של $\frac{N}{2}:1$ ואז ארבעה CMU של $\frac{N}{4}:1$ עד שמגיעים לבעיה שאנחנו מכירים של CMUX 2:1.

בכל פעם הכנסנו את ביט ה MSB של הSELECT להיות ה MUX האחרון בשרשרת ושלחנו קריאה לפונקציה עבור דרגה תחתונה יותר עם ביטי ה SELECT שקיבלנו פחות ביט הMSB

```python
# The bitset counts from left to right, i.g:bitset= 0100 with s=01 will return 1.
def cmux_n_1(bitset,s):
    if len(bitset)==2:
        a=bitset[0]
        b=bitset[1]
        return cmux2_1(a,b,s)
    else:
        midpoint=len(bitset)//2
        bitset1=bitset[:midpoint]
        bitset2=bitset[midpoint:]
        snew=s[1:]
        a=cmux_n_1(bitset1,snew)
        b=cmux_n_1(bitset2,snew)
        return cmux2_1(a,b,s[0])
```

בנינו פונקציה רקורסיבית שמדמה AND TREE באופן דומה לCMUX

```
def and_tree(bitset):
    if len(bitset)==2:
        return integrated_metastability_detection(bitset,"a&b")
    else:
        midpoint=len(bitset)//2
        bitset1=bitset[:midpoint]
        bitset2=bitset[midpoint:]
        newbitset=[]
        newbitset.append(and_tree(bitset1))
        newbitset.append(and_tree(bitset2))
        strnewbitset=''.join(str(num) for num in newbitset)
        return integrated_metastability_detection(strnewbitset,"a&b")
```

אם הביט באורך 2 נחזיר זאת עם כביטוי של AND בודד שהספרייה יודעת להתמודד עם. (אם הוא לא 2 אז נחלק ב2 רקורסיבית ונעשה AND בין התוצאות).

באופן דומה הפונקציה עבור OR TREE הרקורסיבי יהיה:

```
def or_tree(bitset):
    if len(bitset)==2:
        return integrated_metastability_detection(bitset,"a|b")
    else:
        midpoint=len(bitset)//2
        bitset1=bitset[:midpoint]
        bitset2=bitset[midpoint:]
        newbitset=[]
        newbitset.append(or_tree(bitset1))
        newbitset.append(or_tree(bitset2))
        strnewbitset=''.join(str(num) for num in newbitset)
        return integrated_metastability_detection(strnewbitset,"a|b")
```

generate_combinations_indexes מייצרת קומבינציות של ביטסט על פי אינדקסים נתונים. היא מחליפה את הביטים בעמדות המסומנות באינדקסים עם '0' ו-'1' ומייצרת את כל הקומבינציות האפשריות בעזרת פונקציה רקורסיבית פנימית.

```python
def generate_combinations_indexes(bitset, indices):
    # Convert bitset to a list to modify specific indices
    bitset_list = list(bitset)
    combinations = []

    # Recursive helper function to generate combinations
    def generate_helper(current_bitset, idx):
        if idx == len(indices):
            combinations.append(''.join(current_bitset))
            return
        # Get the current index to modify
        current_index = indices[idx]
        # Replace at the current index with '0' and '1', then recurse
        for bit in ['0', '1']:
            current_bitset[current_index] = bit
            generate_helper(current_bitset, idx + 1)

    # Start the recursive generation
    generate_helper(bitset_list, 0)
    return combinations
```

generate_n_choose_k מייצרת את כל הקומבינציות האפשריות של בחירה k מתוך n. היא משתמשת בפונקציית combinations מספריית itertools כדי ליצור את הקומבינציות ומחזירה אותן בפורמט של רשימות.

```python
def generate_n_choose_k(n, k):
    # Generate all combinations of n items taken k at a time without repetition
    items = range(n)  # Items represented by indices starting from 0 to n-1
    all_combinations = list(combinations(items, k))
    # Convert tuples to lists for the output format you requested
    return [list(comb) for comb in all_combinations]
```

compute_all_unions מחשבת את כל האיחודים האפשריים בין קבוצות נתונות. היא עוברת על כל זוגות הקבוצות (כולל כל קבוצה עם עצמה), מחשבת את האיחוד, ומוסיפה אותו לרשימת התוצאות.

```python
def compute_all_unions(sets_list):
    all_unions = []

    # Iterate over each pair of sets (including each set with itself)
    for i in sets_list:
        for j in sets_list:
            # Compute the union and add it to the result list
            union_set = sorted(list(set(i).union(set(j))))
            all_unions.append(union_set)

    return all_unions
```

mask_bitset_with_m מחליפה ביטים בעמדות מסוימות בביטסט עם M. היא מקבלת ביטסט ורשימת אינדקסים, ומחליפה את הביטים בעמדות המסומנות עם 'M'.

```python
def mask_bitset_with_m(bitset, indices):
    # Convert the bitset string to a list of characters for easier manipulation
    bitset_list = list(bitset)

    # Iterate through the indices list and replace the corresponding characters with 'M'
    for index in indices:
        if 0 <= index < len(bitset_list):  # Check if the index is within the bounds of the bitset
            bitset_list[index] = 'M'

    # Convert the list back to a string
    masked_bitset = ''.join(bitset_list)

    return masked_bitset
```

extend_to_next_power_of_two_ones מרחיבה ביטסט למספר הביטים הקרוב ביותר שהוא חזקה של 2 על ידי הוספת '1'. הפונקציה מחפשת את מספר הביטים הקרוב ביותר שהוא חזקה של 2 ומוסיפה '1' לביטסט בהתאם. נשים לב שהוספת ביטי 1 לא משנה את התוצאה כיוון שאנחנו מבצעים AND בין כל השערים.

```python
def extend_to_next_power_of_two_ones(bitset):
    # Determine the current length of the bitset
    current_length = len(bitset)

    # Find the next power of 2 greater than or equal to the length of the b
    if current_length & (current_length - 1) == 0:
        # Already a power of two
        next_power_of_two = current_length
    else:
        next_power_of_two = 2 ** math.ceil(math.log2(current_length))

    # Calculate the number of '1's to add
    number_of_ones_to_add = next_power_of_two - current_length

    # Extend the bitset with '1's
    extended_bitset = bitset + '1' * number_of_ones_to_add

    return extended_bitset
```

extend_to_next_power_of_two_zeros באופן דומה מרחיבה ביטסט למספר הביטים הקרוב ביותר שהוא חזקה של 2 על ידי הוספת '0'. הפונקציה מחפשת את מספר הביטים הקרוב ביותר שהוא חזקה של 2 ומוסיפה '0' לביטסט בהתאם, מתאים להרבה לפני כניסה ל ORTREE כיוון ש0 לא ישפיע.

closest_power_of_2 באופן דומה מחשבת את החזקה הקרובה ביותר של 2 עבור מספר נתון. היא משתמשת בהזזות ביטים כדי למצוא את החזקה הקרובה ביותר של 2 שהיא שווה או גדולה מהמספר הנתון.

circute_comments מבצעת את כל התהליך של זיהוי מטא-סטביליות בעזרת מעגל לוגי, כולל הדפסות של כל שלבי הביניים. היא כוללת יצירת קומבינציות, חישוב איחודים, החלת שערי MUX, שערי AND ושערי OR, ומדפיסה את תוצאות הביניים והתוצאה הסופית.

```python
    ##############
    print("bitsets_to_and=",bitsets_to_and)

    and_res=[]
    for set in bitsets_to_and:
        s=""
        for i in set:
            s+=str(i)
        s=extend_to_next_power_of_two_ones(s)
        and_res.append(and_tree(s))

    ##############
    print("and gates resullts: ",and_res)

    s=""
    for res in and_res:
        s+=str(res)

    s=extend_to_next_power_of_two_zeros(s)
    final_resullt=or_tree(s)

    ###############
    print(f"The final resuult is: {final_resullt}\n")

    return final_resullt
```

```python
def circute_comments(n,k,expr,bitset):
    combinations=generate_n_choose_k(n,k)
    print("The combinations of n choose k are:",combinations)
    unions=compute_all_unions(combinations)
    print("The all posible unions are: ", unions)
    m_unions=[]
    for union in unions:
        m_unions.append(mask_bitset_with_m(bitset,union))
    print("The all union posibillitys with m are:",m_unions)
    mux_res=[]
    list_to_mux=[]
    bits_to_mux=""
    s=""
    counter=0
    for union in m_unions:
        ###########
        print(f"mux number {counter+1}")
        list_to_mux=[]
        s=""
        bits_to_mux=""
        list_to_mux=generate_combinations_indexes(union,unions[counter])
        ############
        print("list to mux= ",list_to_mux)
        for set in list_to_mux:
            bits_to_mux+=str(integrated_metastability_detection(set,expr))
        ############
        print("bits to mux= ",bits_to_mux)
        for i in unions[counter]:
            s+=str(bitset[i])
        ############
        print("s= ",s)
        mux_res.append(cmux_n_1(bits_to_mux,s))

        counter+=1

    ###########
    print("mux res= ",mux_res)

    and_gate_size=len(combinations)
    bitsets_to_and=[]
    for i in range(0,len(mux_res),and_gate_size):
        bitsets_to_and.append(mux_res[i:i+and_gate_size])

    ##############
    print("bitsets_to_and=",bitsets_to_and)
```

circuit דומה ל-circute_comments אך ללא הדפסות של שלבי הביניים. היא מבצעת את התהליך באופן יעיל יותר ומחזירה את התוצאה הסופית בלבד.

gates מחשבת את מספר השערים הנדרשים לביצוע מעגל לוגי נתון. היא סופרת את מספר שערי AND, OR ו-NOT הנדרשים, ואת מספר השערים הנוספים הנדרשים עבור הביטויים הלוגיים, גם בAND וגם בOR עבור בנייה למספר בן 2 בחזקת n ביטים יהיו n שלבים, בסופו של דבר מספר השערים בנוי מסדרה הנדסית $1 + 2 + 4 + 8 \ldots + 2^n$ כך שעבור עץ שנכנסים אליו $8 = 2^n$ ביטים נקבל סדרה של $1 + 2 + 4$ שערים.
אותו עקרון עבור הOR TREE ועבור הCMUX, רק צריך לשים לב שאת ה AND TREE נכפיל במספר הN בחר K שלנו כי יש מספר כזה של AND TREE.

```python
def gates(n,k):
    num_and=0
    num_or=0
    num_not=0
    combinations_num=math.comb(n, k)
    size_of_tree=closest_power_of_2(combinations_num)
    for i in range (0,int(math.log(size_of_tree,2))-1):
        num_and=num_and+2**i
        num_or=num_or+2**i
    num_and=num_and*combinations_num
    x=generate_n_choose_k(n,k)
    y=compute_all_unions(x)
    mux_2=0
    expressions=0
    for i in y:
        expressions=expressions+2**len(i)
        for j in range (0,len(i)-1):
            mux_2=mux_2+2**j

    num_or=num_or+mux_2*3
    num_and=num_and+mux_2*3
    num_not=mux_2

    print(f"Total:\nand gates: {num_and}\nor gates: {num_or}\nnot gates: {num_not}\nalso the number of gates
```

gates_no_comments

דומה ל-gates אך מחזירה את סך כל השערים הנדרשים ללא הדפסות של מידע ביניים. היא מחשבת את מספר השערים הנדרשים עבור מעגל לוגי נתון ומחזירה את הסכום הכולל.

letancy מחשבת את ההשהייה הכוללת של מעגל לוגי נתון, בהתבסס על פרמטרים n ו-k. הפונקציה מתחילה בהגדרת משתנים המייצגים את ההשהייה של שערי AND, OR, ו-NOT.
הפונקציה מגדירה משתנה למעקב אחרי ההשהייה של השערים מסוג MUX-2.
הפונקציה משתמשת ב-n ו-k כדי לחשב את מספר הקומבינציות האפשריות של בחירה k מתוך n.

הפונקציה יוצרת את הקומבינציות בעזרת generate_n_choose_k, ואת האיחודים של הקומבינציות בעזרת compute_all_unions. לאחר מכן, הפונקציה מחשבת את מספר השערים מסוג MUX-2 הנדרשים, כאשר חישוב זה הוא ההשהייה של הCMUX הכי גדול שנבנה במעגל ובו המסלול הכי ארוך.
כאשר המסלול הכי ארוך נקבע מאיחוד של קבוצות (אם האיחוד יוצר הרבה אופציות של ביטים שיכולים להיות M אזי הכניסה למוקס תהיה גדולה יותר ובהתאם המוקס גדול יותר).
הפונקציה מחשבת ומוסיפה לזה את מספר ההשהייה של שערי AND ו-OR בהתבסס על גודל העץ וכמה שכבות יש לו ומעדכנת את ההשהייה הכוללת בהתאם.

בסיום, הפונקציה מחזירה את ההשהייה הכוללת במיקרו-שניות, המבוססת על סיכום ההשהיות של כל השערים הנדרשים במעגל הלוגי. כל ההשהיות מבוססות על הערכות של זמני ההשהייה של השערים השונים, כאשר הנחנו דיליי אופייני של שערים על ידי : כל שער AND ו-OR מתווסף עם השהייה של 50 פיקו-שניות, וכל שער NOT מתווסף עם השהייה של 20 פיקו-שניות.

```
def letancy(n,k):
    or_way=0
    and_way=0
    not_way=0
    mux_2_way=0
    combinations_num=math.comb(n, k)
    size_of_tree=closest_power_of_2(combinations_num)
    or_way=int(math.log(size_of_tree,2))
    and_way=int(math.log(size_of_tree,2))
    x=generate_n_choose_k(n,k)
    y=compute_all_unions(x)
    lenths=[len(i) for i in y ]
    mux_2_way=max(lenths)
    not_way=mux_2_way
    or_way=or_way+mux_2_way*3
    and_way=and_way+mux_2_way
    return and_way*50+or_way*50+not_way*20
```

plot_graphs

מציירת גרפים של ההשהייה ומספר השערים הנדרשים עבור מעגלים לוגיים בקשרים שונים של n ו-k. היא יוצרת שני גרפים: אחד עבור ההשהייה (בפיקושניות) ואחד עבור מספר השערים.

plot_heatmaps

מציירת מפות חום של ההשהייה ומספר השערים הנדרשים עבור טווחי ערכים של n ו-k. היא משתמשת במערך דו-ממדי לאחסון ההשהייה ומספר השערים ומציירת את המפות עם צבעים המייצגים את ההשהייה ומספר השערים.

-דוגמת הרצה-

נכניס תחילה את מספר הביטים n במעגל ואת K מספר הביטי מטאסטיבליות

```
C:\Users\yaniv\AppData\Local\Programs\Python\Python39\python.exe
Enter the size of the bit set
5
Enter the size of the metastabillic bits that should come in the input
3
Please press 1 if you want to see a resullt for a spesific bitset
Press 2 if you want to see the resullts for an every combination of bitset that can be in the size that you have choosen
Press 3 for checking letancy and number of gates on range of 1-k in n
Press 4 for checking the all range betwwn 1 to k and 1 to n when k<=n
```

כעת ישנן 4 אופציות:

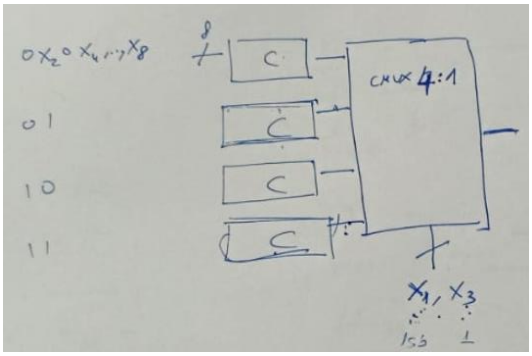
אופציה 1

מאפשרת לבדוק תוצאה עבור אופציה בודדת של ביטים.

נתבקש להכניס את הביטוי הלוגי והשמה בודדת (לדוגמה עבור a|b|c וכניסה M10)

```
please enter a boolean expression, use "&" for "and","|" for "or","~" for "not"
For the expression use the bit from 'a' to 'z for example "(~a&b)|c
a|b|c
Please enter the spesific bitset you want to check it on, for example: 'M10'
M10
```

נקבל תוצאה מהמעגל ומהפונקציה f_m (משווה האם התוצאות שוות), כמו כן מראה כמה שערי and כמה שערי or וכמה שערי not המעגל דרש כדי לקיים את דרישות הבנייה. יש לשים לב שלתוך כל CMUX נכנס המימוש של הביטוי הלוגי שהוזן, כאשר כל C בציור זה מימוש של המעגל עבור a|b|c באופן הבא:



הקוד מחשב את מספר שערי הC הדרושים, יש לקחת בחשבון ולהוסיף את מספר השערים בבנייה C ולהכפיל (במקרה זה להכפיל ב60 כי יש בבנייה 60 שערי C)

```
As we can see the resullt from the circute is 1 when the resullt from the res function is 1 in this case they
are equal
Total:
and gates: 66
or gates: 64
not gates: 21
also the number of gates that are nedded for the expression are multipled by 60
If we guess that and gate letancy= 50 PicoSeconds, or gate letancy= 50 PicoSeconds,
not gate latency= 20 Pico Seconds
So the total circute latency is 860 PicoSeconds + the latency of the expressions boolean circute
```

כמו כן הקוד מחשב את Latency (הנחנו דיליי אופייני שער AND ו-OR מתווסף עם השהייה של 50 פיקו-שניות, שער NOT השהייה של 20 פיקו-שניות.

ויש להוסיף לזה את Latency שבשער C (מימוש הביטוי הבוליאני).

אופציה 2

גם כאן נכניס תחילה את מספר הביטיחn במעגל ואת K מספר הביטי מטאסטיבליות. אך פה יחושבו כל האופציות השונות של ההשמות לn והרצת המעגל עבור כל אחת מהן.

```
please enter a boolean expression, use "&" for "and","|" for "or","~" for "not"
For the expression use the bit from 'a' to 'z for example "(~a&b)|c
a|b|c
```

יש התייחסות לכל מקרה, האם במקרה זה מספר הביטים במטאסטבילים גדולה\קטנה\שווה לK. לבסוף לאחר חישוב כל התוצאות נקבל כמה תוצאות היו זהות וכמה שונות מהפונקציה f_m (במקרה זה כל התוצאות היו זהות).

```
C:\Users\yaniv\AppData\Local\Programs\Python\Python39\python.exe
The number of the M bits is equal than k=2
MM1 -> Result from res function: 1
MM1 -> Result from Cmux circute: 1
True
The number of the M bits is bigger than k=2
MMM -> Result from res function: M
MMM -> Result from Cmux circute: M
True
Same results count: 27
Different results count: 0
```

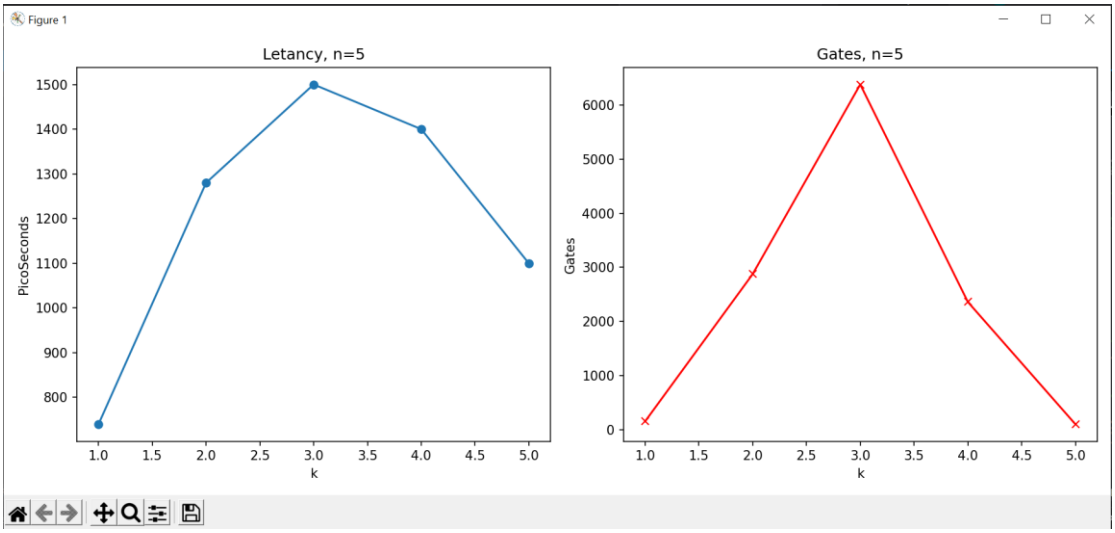
ובנוסף נקבל מידע אודות מספר השערים והlatency באופן דומה לאופציה 1.

```
also the number of gates that are nedded for the expression are multipled by 60
If we guess that and gate letancy= 50 PicoSeconds, or gate letancy= 50 PicoSeconds,
not gate latency= 20 Pico Seconds
So the total circute latency is 860 PicoSeconds + the latency of the expressions boolean circute
```

אופציה 3

אופציה זו מיועדת להדגים ולנתח את הזמן הכולל של תגובת המעגל ואת מספר השערים הנדרשים עבור קלטים בגודל שונה של k תוך שמירה על גודל n. פונקציה זו משתמשת בגרף כדי להדגים את התלות בין גודל k לבין זמן התגובה ומספר השערים. זה מאפשר לראות איך הביצועים של המעגל משתנים כתלות במספר הקומבינציות האפשריות של סטים שנבחרים.

לדוגמה עבור n=5 וk=2



אופציה 4

אופציה זו מרחיבה את הניתוח ומדגימה את התלות של זמן התגובה ומספר השערים בכל הטווחים האפשריים של גודלי k ו-n. היא מציגה מפת חום של זמן תגובה ושערים עבור כל זוג של n ו-k. זה מאפשר תובנה מעמיקה יותר על איך הגודל השונה של הקלטים והסטים משפיע על הביצועים של המעגל, ומספק מידע חיוני לתכנון יעיל ומותאם.