Application of Boolean Algebra and Bitwise Masking for Player Movement State Management in Unity Engine

Nadia Aulia Syafarani - 13525122
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: auliaunad@gmail.com , 13525122@std.stei.itb.ac.id

***Abstract*—State management is a crucial part in developing video games in order to maintain gameplay consistency and responsiveness. However, implementation of player state management is usually within nested conditional logic that relies more on intuition rather than mathematical validation. By using bitwise masking to store several states in one byte, this paper aims to apply the validation process of Boolean algebra to state management. Using C# inside Unity engine, the proposed approach uses verifiable state management with truth tables and Karnaugh maps in order to demonstrate a mathematically verifiable management system (*Abstract*)**

Keywords—boolean algebra; bitwise; game development; state management; Unity engine (key words)

I. Introduction

The game industry is one of the fastest-growing parts of software, and by 2025 the global market was worth about $201.6 billion [1]. Platformer games specifically have stayed popular for years, from classic games like Super Mario Bros to newer indie hits such as Celeste and Hollow Knight. Platformer is a genre of game in which the player controls a character to move and maneuver across platforms, requiring various different states of movements to get the player from point A to point B. In order to manage these states, game developers use a tool known as state management.

In game engineering, state management often utilizes a tool of abstraction called a Finite State Machine. A finite state machine (or usually referred to as a finite state automaton) is a computation model consisting of states, initial state, inputs, transitions, and outputs that is used as a blueprint for a system that can only be one state at a time [2]. In the context of player movement, this can be used to map out the states in which a character can be in. Each state is used to represent what type of movement state the character is in (e.g. grounded, jumping, and dashing) and transitions are triggered based on the player's input. However, in practice, FSMs are usually implemented using numerous boolean variables and nested within multiple conditions. While this approach is fine within a simplistic application of movements in a game, when the complexity increases the issue of logical validation arises. Without any formal guarantees, a player can be in two conflicting states without any mathematical mechanism to prevent it. One way to solve this lack of formal verification is to look at classic mathematical theories.

Boolean algebra, founded by George Boole in the 19th century, is a branch of discrete mathematics that deals with logical conditions represented as binary values (0 and 1) [3]. Using the three fundamental operations (AND, OR, and NOT) allow us to calculate and verify the combinations of true and false conditions. In the context of programming these operations can be directly used with bitwise operations, which apply Boolean logic to bits of an integer. By representing each player state as a single bit within a byte variable, we can check multiple states within a single structure. This way, the validation of conflicting states can be verified mathematically with precise calculations. Despite how common FSMs are in game development, most implementations rely on developer's intuition rather than solid, verifiable mathematical guarantee to ensure state correctness. There's no widespread practice of formally verifying player state using Boolean algebra, especially in the context of Indie game development.

With thousands of games being released each year, a demand for a game engine that is powerful but also accessible is on the rise. This is where Unity comes in. It's one of the most widely used game engines out there, and games like Genshin Impact show that it can handle large-scale projects just fine. At the same time, it's especially popular among indie developers, with titles like Among Us and Hollow Knight being good examples. If we take a look at Hollow Knight's movement mechanic, we will find some set rules in the player states. For example, in Hollow Knight when we are currently jumping, it is impossible for us to be in a grounded state, proving that some of the player states are mutually exclusive by design. This ensures relative accuracy to the physics that plays in order to make the movement feels more natural, and also to uphold consistency in the gameplay. While this consistency is crucial for high-quality gameplay, translating these mutually exclusive design rules into a script remains a persistent challenge for independent creators.

To bridge this gap, this paper proposes the application of Boolean algebra and bitwise masking as a formal approach to represent and verify player movement states in Unity Engine. By encoding all movement states into a single byte variable, each bit representing one condition, we apply the

mathematical properties of Boolean algebra to formally analyze the state space. Through truth table analysis, Karnaugh map minimization, and Boolean algebraic proof, we demonstrate that the implementation is guaranteed to never enter a contradictory state. The case study is based on a 2D platformer movement system implemented in Unity Engine.

II. Theoretical Background

A. Boolean Algebra

To understand Boolean Algebra formally, first we need to know what Boolean Algebra consists of. Before going into the structures, there are some variables that need to be defined, which are"

1. B, which is the set of elements
2. $+$ to represent OR operation (also known as sum)
3. $\cdot$ to represent AND operation (also known as product)
4. $'$ to represent NOT operation (also known as complement).

The structure of Boolean Algebra is as follows:

$$<B, +, \cdot, ', 0, 1> \quad (1)$$

In the structure shown in (1), 0 and 1 represent two distinct unique elements within the bounded set B ($0,1 \in B$) [3]. For this framework to be valid mathematically, every element a, b, $c \in B$ should satisfy a set of fundamental axioms as shown below:

1. Identity
   a. $a + 0 = a$
   b. $a \cdot 1 = a$
2. Commutative
   a. $a + b = b + a$
   b. $a \cdot b = b \cdot a$
3. Complement

   For every a $\in B$ exists a unique element a' $\in B$ such that

   a. $a + a' = 1$
   b. $a \cdot a' = 0$
4. Distributive
   a. $a \cdot (b + c) = (a \cdot b) + (a \cdot c)$
   b. $a + (b \cdot c) = (a + b) \cdot (a + c)$
5. Associative
   a. $a + (b + c) = (a + b) + c$
   b. $a \cdot (b \cdot c) = (a \cdot b) \cdot c$

Boolean algebra operates binary values, in this case set B with 0 and 1 elements, using AND, OR, and NOT operators. These are the three fundamental operators of Boolean algebra and will result in binary values according to the conventions shown below.

a	b	$a \cdot b$
0	0	0
0	1	0
1	0	0
1	1	1

Fig. 1. Product operator conventions. Adapted from [3]

a	b	$a + b$
0	0	0
0	1	1
1	0	1
1	1	1

Fig. 2. Sum operator conventions. Adapted from [3]

a	a'
0	1
1	0

Fig. 3. Complement operator conventions. Adapted from [3]

By combining these operators with variables, we can create a logical expression known as a Boolean expression. These expressions can be used to describe certain conditions, for example in programming concepts it is used in if-else branching. Boolean expressions are used within a set of laws as shown below.

TABLE I. Boolean Algebra Laws

No.	***Law***	***Expression***
1.	Identity Law	(i) $a + 0 = a$ (ii) $a \cdot 1 = a$
2.	Idempotent Law	(i) $a + a = a$ (ii) $a \cdot a = a$
3.	Complement Law	(i) $a + a' = 1$ (ii) $a \cdot a' = 0$
4.	Dominance Law	(i) $a \cdot 0 = 0$ (ii) $a + 1 = 1$
5.	Involution Law	(i) $(a')' = a$
6.	Absorption Law	(i) $a + ab = a$ (ii) $a(a + b) = a$
7.	Commutative Law	(i) $a + b = b + a$ (ii) $ab = ba$
8.	Associative Law	(i) $a + (b + c) = (a + b) + c$ (ii) $a(bc) = (ab)c$
9.	Distributive Law	(i) $a + bc = (a + b)(a + c)$ (ii) $a(b + c) = ab + ac$
10.	De Morgan's Law	(i) $(a + b)' = a'b'$ (ii) $(ab)' = a' + b'$
11.	0/1 Law	(i) 0' = 1 (ii) 1' = 0

II. Adapted from [3]

A Boolean function is one of the usage of Boolean expressions, returning either 0 or 1 depending on the values assigned to the variables. Each variable within a Boolean function is known as a literal [3]. There are multiple ways to write out Boolean functions, often known as canonical form,

with the two methods being Sum of Product (SOP) and Product of Sum (POS).

When written in canonical form, each term will include all of the variables in the function. When written in SOP form, each term is known as a minterm, where each minterm is made up of the AND of all variables. When written in POS form, each term is known as a maxterm, where each maxterm is made up of the OR of all variables [3]. For minterms, every variable with the value 1 is represented with the non-complement form. For maxterms, every variable with the value 0 is represented with the non-complement form. A truth table can be used to easily show these relationships.

		Minterm		*Maxterm*	
x	*y*	Suku	Lambang	Suku	Lambang
0	0	*x'y'*	m_0	$x + y$	M_0
0	1	*x'y*	m_1	$x + y'$	M_1
1	0	*xy'*	m_2	$x' + y$	M_2
1	1	*x y*	m_3	$x' + y'$	M_3

Fig. 4. Examples of minterms and maxterms for two variables. Adapted from [3]

B. Logic Minimization using Karnaugh Maps

In the application of Boolean algebra, a minimized Boolean function results in a simpler logic circuit, making it more optimized. In order to minimize a function, we have to find an equivalent function in terms of logic, but with the least possible literals or number of operations. One way to do it is using the Karnaugh Map Method.

			y = 0	*y* = 1
m_0	m_1	*x* = 0	*x'y'*	*x'y*
m_2	m_3	*x* = 1	*xy'*	*xy*

Fig. 5. Two-variable K-map. Adapted from [6]

Maurice Karnaugh made a discovery in 1953, where he created a map comprising of adjacent cells, each representing a minterm. This is referred to as a Karnaugh Map or K-map [6]. These maps can be drawn for two or more variables. Two variable K-map will comprise of four cells, a three-variable K-map will have six cells, a four variable K-map will contain eight cells and so on. Adjacent cells in a K-map should be able to differ from each other by just one literal [6]. In a four variable map, the cells arrangement will be in the form of 00 → 01→ 11 → 10.

m_0	m_1	m_3	m_2
m_4	m_5	m_7	m_6

x \ *yz*	00	01	11	10
0	*x'y'z'*	*x'y'z*	*x'yz*	*x'yz'*
1	*xy'z'*	*xy'z*	*xyz*	*xyz'*

Fig. 6. Three-variable K-map. Adapted from [6]

m_0	m_1	m_3	m_2
m_4	m_5	m_7	m_6
m_{12}	m_{13}	m_{15}	m_{14}
m_8	m_9	m_{11}	m_{10}

wx \ *yz*	00	01	11	10
00	*w'x'y'z'*	*w'x'y'z*	*w'x'yz*	*w'x'yz'*
01	*w'xy'z'*	*w'xy'z*	*w'xyz*	*w'xyz'*
11	*wxy'z'*	*wxy'z*	*wxyz*	*wxyz'*
10	*wx'y'z'*	*wx'y'z*	*wx'yz*	*wx'yz'*

Fig. 7. Four-variable K-map. Adapted from [6]

The Boolean function minimization or reduction begins with the process of assigning some variables to the map. This assignment could be based either on the truth table or the Boolean function itself. The cells which represent the minterms, are assigned the output of 1, while the others are assigned 0 [6]. The adjacent cells, which have the value of 1, are combined into groups that have the size of a power of two, like pairs, quad, octet, etc.

x \ *yz*	00	01	11	10
0	0	0	1	1
1	1	0	1	1

$xz' + y$

Fig. 8. Example for filling a K-map with the function $F(x, y, z) = xz' + y$. Adapted from [6]

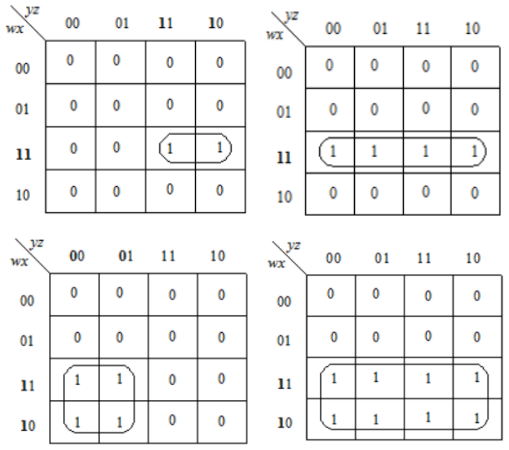

Fig. 9. Pair, quad, and octet grouping. Adapted from [6]

K-maps can also contain a "don't-care" state, in which a variable isn't relevant to the current function, so the cell is assigned an output of X. When dealing with these states, we first assume the value of X as 1 and attempt to group them with known values. Then, when everything has been grouped, we assume the value of the ungrouped Xs as 0 [6].

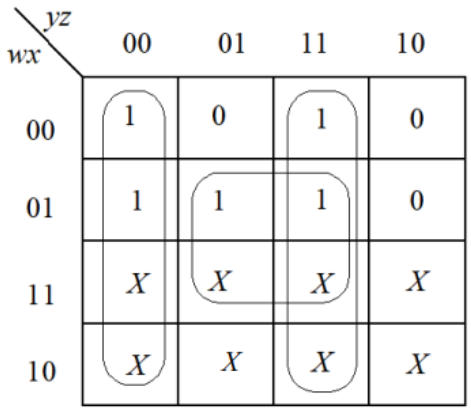

Fig. 10. Grouping with don't-care states. Adapted from [6]

C. Bitwise Operations

Boolean algebra is a good way to understand logic in the abstract, but bitwise operations are what computers actually execute at the hardware level. In a CPU, those operations run inside the ALU, so they are fast and direct. That matters because they avoid some of the overhead that comes with high-level branching logic.

The basic bitwise operators are AND (&), OR (|), XOR (^), and NOT (~). Much like Boolean algebra, these operators work in a way in which AND returns 1 only when both bits are 1, OR returns 1 when at least one bit is 1, XOR returns 1 when the bits differ (i.e. when one is 0 and the other is 1), and NOT flips every bit [4], [5].

X	Y	X & Y	X | Y	X ^ Y
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Fig. 11. Truth tables of bitwise operators. Adapted from [4].

```c
#include <stdio.h>

int main()
{

    // a = 5 (00000101 in 8-bit binary)
    // b = 9 (00001001 in 8-bit binary)
    unsigned int a = 5, b = 9;

    // The result is 00000001
    printf("a&b = %u\n", a & b);

    // The result is 00001101
    printf("a|b = %u\n", a | b);

    // The result is 00001100
    printf("a^b = %u\n", a ^ b);

    // The result is 11111111111111111111111111111010 (assuming 32-bit
unsigned int)
    printf("~a = %u\n", a = ~a);
```

Fig. 12. Examples of bitwise AND, OR, XOR, and NOT operations. Adapted from [4]

Another useful bitwise operator found in C# are the shift operators. Left shift (<<) moves bits toward higher positions, generally used for multiplication by powers of two, and right shift (>>) moves them toward lower positions, generally used for divisions by powers of two [5]. They can also be used to isolate specific bits depending on it's usage.

```c
uint x = 0b_1100_1001_0000_0000_0000_0000_0001_0001;
Console.WriteLine($"Before: {Convert.ToString(x, toBase: 2)}");

uint y = x << 4;
Console.WriteLine($"After:  {Convert.ToString(y, toBase: 2)}");
// Output:
// Before: 11001001000000000000000000010001
// After:  10010000000000000000000100010000
```

```c
uint x = 0b_1001;
Console.WriteLine($"Before: {Convert.ToString(x, toBase: 2), 4}");

uint y = x >> 2;
Console.WriteLine($"After:  {Convert.ToString(y, toBase: 2).PadLeft(4, '0'), 4}");
// Output:
// Before: 1001
// After:  0010
```

Fig. 13. Left-shift and right-shift operations. Adapted from [5]

One of the biggest advantages of bitwise operations is that they allow storing several true/false conditions in a single byte. Using this information, we can use each bit to represent a different flag, making it possible to turn a flag on with OR, turn it off with AND together with NOT, and check whether it is set with AND [4], [5]. This pattern is called bitwise masking.

TABLE II. EXAMPLE OF BIT MASKING FOR PLAYER MOVEMENT STATES

Bit Position	*State Flag*	*Bit Mask*
0	GROUNDED	0000 0001
1	RUN	0000 0010
2	JUMP	0000 0100

TABLE III. EXAMPLE OF BITWISE OPERATORS FOR STATE FLAGS

Action	*State Before*	*Expression*	*State After*
Player starts running	00000001	state = state | RUN	00000011
Player jumps	00000011	state &= ~GROUNDED state = state | JUMP	00000110
Player lands	00001010	state = state & ~JUMP state = state | GROUNDED	00000011

D. Movement States in Platformer Games

In standard discussions of game-engine architecture, character movement is often modeled using only two fundamental states: grounded motion and jumping. These two states are sufficient to describe the basic effects of gravity on an object resting on a surface and the application of an initial

upward impulse that temporarily counteracts gravity. However, for contemporary platform games, this simplified framework is often not enough when the objective is to achieve smooth and responsive control.

To improve game feel, designers typically add three more states: dashing, wall-sliding, and hitstop. Together, these five states form a practical set of movement conditions that combine ordinary physics with deliberate interruptions to improve control. In this framework:

1. Grounded state is the base condition in which the character is in contact with the ground. It allows normal horizontal movement and enables jump input to produce an upward impulse.
2. Jumping state occurs when the character is airborne after an upward force is applied. This state reduces some lateral control and changes how air resistance affects motion.
3. Dashing state is a brief burst of high-speed horizontal movement. It temporarily overrides normal gravity behavior so the character can keep a fast, straight path.
4. Wall-sliding state happens when the character is airborne, touching a wall, and pushing toward it. This state adds controlled drag that slows the fall and helps set up a wall jump.
5. Hitstop state pauses movement and animation for a few frames, generally when receiving damage. Its purpose is to emphasize impact and make important actions feel more distinct.

From a systems perspective, these states form a complete set of movement modes that balance physical rules with intentional interruptions for clearer player feedback and better control.

III. System Design

Before implementing the system into the Unity Engine, we need to conceptualize the design of the state management. In order to make sure the system is mathematically verifiable, a clear system design consisting of state definitions, rules, and guard functions needs to be established first and foremost.

A. State Definitions

The first step is defining the states of which a player can go through. Taking user experience and game feel into consideration based on the established theoretical background, we will map the movements into 5 separate variables using the binary set of $B = \{0, 1\}$. The binary representation of these states are as follows.

TABLE IV. Binary encoding of player states

Variable	*State Flag*	*Value*	*Condition*
G	Grounded	1	1 if the player is touching the ground
		0	Player is airborne
J	Jumping	1	Player is accelerating upwards
		0	Any other conditions
D	Dashing	1	Player is in a burst of high speed horizontal movement
		0	Any other conditions
W	Wallsliding	1	Player is sticking and sliding on a wall
		0	Any other conditions
H	Hitstop	1	Player is frozen for a moment to emphasize impact
		0	Any other conditions

TABLE V.

TABLE VI. Binary mask of each player state

Bit Position	*Variable*	*Bit Mask*	*Decimal Value*
0	Grounded	0b00000001	1
1	Jumping	0b00000010	2
2	Dashing	0b00000100	4
3	Wallsliding	0b00001000	8
4	Hitstop	0b00010000	16

B. Design Constraints

In order to effectively design a system in which impossible combinations cannot be performed, there needs to be a design constraint to prevent conflicting states from overlapping. According to Boolean algebra law, contradicting states that can't be active at the same time should adhere to the Complement Law ($ab = 0$). By referring to this law, we can create a set of rules on movement based on the real-life constraints of each movement. The formulated rules are as follows:

1. A player cannot be on the ground and also airborne at the same time. As shown in the equation below.
$$GJ = 0 \quad (2)$$
2. A character cannot be and the ground and also sticking to a wall at the same time. As shown in the equation below.
$$GW = 0 \quad (3)$$
3. A player when pushing against a wall cancels out the force needed to perform a vertical jump. As shown in the equation below.
$$JW = 0 \quad (4)$$

4. A player when doing a dash will cancel the action of pushing against a wall. As shown in the equation below.

$$DW = 0 \qquad (5)$$

Based on the 4 rules in (2), (3), (4), and (5) we can figure out the valid bytes. With 5 bits of different state representation, the total number of representation is $2^5 = 32$. By using a truth table, we can calculate the number of valid bytes.

State	G	J	D	W	H	Minterm	Violation	Output	Status
0	0	0	0	0	0	G'J'D'W'H'	-	0	VALID
1	1	0	0	0	0	GJ'D'W'H'	-	0	VALID
2	0	1	0	0	0	G'JD'W'H'	-	0	VALID
3	1	1	0	0	0	GJD'W'H'	Rule 1 (GJ)	1	INVALID
4	0	0	1	0	0	G'J'DW'H'	-	0	VALID
5	1	0	1	0	0	GJ'DW'H'	-	0	VALID
6	0	1	1	0	0	G'JDW'H'	-	0	VALID
7	1	1	1	0	0	GJDW'H'	Rule 1 (GJ)	1	INVALID
8	0	0	0	1	0	G'J'D'WH'	-	0	VALID
9	1	0	0	1	0	GJ'D'WH'	Rule 2 (GW)	1	INVALID
10	0	1	0	1	0	G'JD'WH'	Rule 3 (JW)	1	INVALID
11	1	1	0	1	0	GJD'WH'	Rule 1, 2, 3	1	INVALID
12	0	0	1	1	0	G'J'DWH'	Rule 4 (DW)	1	INVALID
13	1	0	1	1	0	GJ'DWH'	Rule 2, 4	1	INVALID
14	0	1	1	1	0	G'JDWH'	Rule 3, 4	1	INVALID
15	1	1	1	1	0	GJDWH'	Rule 1, 2, 3, 4	1	INVALID
16	0	0	0	0	1	G'J'D'W'H	-	0	VALID
17	1	0	0	0	1	GJ'D'W'H	-	0	VALID
18	0	1	0	0	1	G'JD'W'H	-	0	VALID
19	1	1	0	0	1	GJD'W'H	Rule 1 (GJ)	1	INVALID
20	0	0	1	0	1	G'J'DW'H	-	0	VALID
21	1	0	1	0	1	GJ'DW'H	-	0	VALID
22	0	1	1	0	1	G'JDW'H	-	0	VALID
23	1	1	1	0	1	GJDW'H	Rule 1 (GJ)	1	INVALID
24	0	0	0	1	1	G'J'D'WH	-	0	VALID
25	1	0	0	1	1	GJ'D'WH	Rule 2 (GW)	1	INVALID
26	0	1	0	1	1	G'JD'WH	Rule 3 (JW)	1	INVALID
27	1	1	0	1	1	GJD'WH	Rule 1, 2, 3	1	INVALID
28	0	0	1	1	1	G'J'DWH	Rule 4 (DW)	1	INVALID
29	1	0	1	1	1	GJ'DWH	Rule 2, 4	1	INVALID
30	0	1	1	1	1	G'JDWH	Rule 3, 4	1	INVALID
31	1	1	1	1	1	GJDWH	Rule 1, 2, 3, 4	1	INVALID

Fig. 14. Truth table of all the possible byte representation

From the truth table above we can see that there are 14 valid states, them being 0, 1, 2, 4, 5, 6, 8, 16, 17, 18, 20, 21, 22, and 24. This will be the guideline during implementation, as a verification tool in which every state that is possible during runtime are one of the valid states mentioned above.

IV. Analysis and Minimization

After we've defined the restraints we have to abide by, now we can start designing the guard conditions that will govern how a player enters and exits a state. These guard conditions will be proven using truth tables and minimized with K-Maps before being implemented.

A. canJump Condition

A player can enter the jump state if they are currently on the ground, they are not currently jumping, and they are not currently dashing. Based on those conditions a Boolean function and its truth table can be formed as seen below

$$canJump(G, J, D) = GJ'D' \qquad (6)$$

G	J	D	GJ'D'
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

Fig. 15. Truth table of canJump function

G \ JD	00	01	11	10
0	0	0	0	0
1	1	0	0	0

Fig. 16. K-Map of canJump function

Based on the truth table above we can conclude that there is only one combination in which a player can jump, which is $G = 1$, $J = 0$, and $D = 0$. Based on the K-Map we can see that there is no way to group it forming a pair, quad, nor octet. Therefore, the function is already the most simple it can be.

B. canWallJump Condition

A player can enter the wall jump state if they are currently not wall sliding and not currently on the ground. Based on those conditions a Boolean function and its truth table can be formed as seen below

$$canWallJump(W, G) = WG' \qquad (7)$$

W	G	WG'
0	0	0
0	1	0
1	0	1
1	1	0

Fig. 17. Truth table of canWallJump function

W \ G	0	1
0	0	0
1	1	0

Fig. 18. K-Map of canWallJump function

Based on the truth table above we can conclude that there is only one combination in which a player can jump, which is $W = 1$ and $G = 0$. Based on the K-Map we can see that there is also no pair to be formed so the function is already the most simple it can be.

C. *canDash Condition*

A player can enter the wall jump state if they are currently not dashing and not frozen in hitstop. Based on those conditions a Boolean function and its truth table can be formed as seen below

$$canDash(D, H) = D'H' \quad (8)$$

D	H	D'H'
0	0	1
0	1	0
1	0	0
1	1	0

Fig. 19. Truth table of canDash function

D \ H	0	1
0	1	0
1	0	0

Fig. 20. K-Map of canDash function

Based on the truth table above we can conclude that there is only one combination in which a player can jump, which is $D = 0$ and $H = 0$. Based on the K-Map we can see that there is also no pair to be formed so the function is already the most simple it can be..

D. *wallSlide Condition*

For wallSlide state we use an outside boolean variable that is calculated within the Unity Engine. These variables are T which represents a player touching the wall and M which represent a player inputting the move button towards a wall. A player can enter the wallslide state when they are touching the wall, moving towards the wall, not currently grounded, and not currently dashing. Based on those conditions a Boolean function can be formed as seen below

$$canWallSlide(T, M, G, D) = TMG'D' \quad (9)$$

E. *Validation Condition*

By making a K-map of every minterm in the 5-variable states, we can make a validation function that is used to check if a state is valid or not. This is to detect bugs or accidental overlap that may happen during runtime.

GJ \ DWH	000	001	011	010	110	111	101	100
00	0	0	0	0	1	1	0	0
01	0	0	1	1	1	1	0	0
11	1	1	1	1	1	1	1	1
10	0	0	1	1	1	1	0	0

Fig. 21. K-Map of validation function

Using the K-map method in which we group the cells containing the value of 1, we can find that the 18 invalid states can be minimized in the form of SOP, creating a validation function as follows:

$$valid(G, J, D, W, H) = GJ + GW + JW + DW \quad (10)$$

V. IMPLEMENTATION

Based on the analysis on section IV, this section will detail on the implementation of player movement states using Boolean algebra logic and bitwise masking. This program is coded using C# in the Unity 2D game engine with the hierarchy setup as seen below

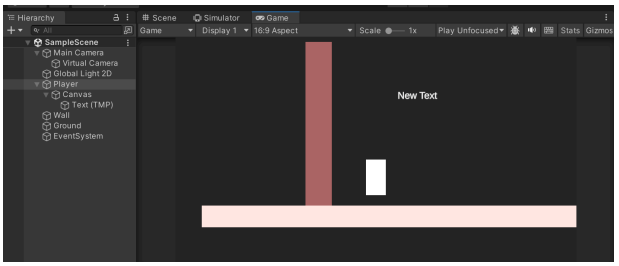

Fig. 22. Unity scene setup

This is a simple scene setup with a wall, ground, player, and text using TextMeshPro that will be used as the real-time HUD for checking the validity of the bitwise masking. In the player is attached a script called PlayerController, in which the system design has been implemented accordingly. The code that follows is a direct translation of the previous mathematical functions and analysis.

A. *State Declarations*

```
// STATE DEFINITIONS
10 references
private const byte STATE_GROUNDED    = 0b00000001; // bit 0
10 references
private const byte STATE_JUMPING     = 0b00000010; // bit 1
9 references
private const byte STATE_DASHING     = 0b00000100; // bit 2
7 references
private const byte STATE_WALLSLIDING = 0b00001000; // bit 3
5 references
private const byte STATE_HITSTOP     = 0b00010000; // bit 4

52 references
private byte state = STATE_GROUNDED;
```

Fig. 23. State declaration variables

This is the implementation of table VI in section III. Here each constant represents a state, each assigned a bit position inside a byte. The state STATE_GROUNDED is used as the basic state which will be operated later using bitwise operators when changing states.

B. *Jump Logic*

```
void HandleJump()
{
    // GROUNDED AND NOT JUMPING AND NOT DASHING
    bool canJump = (state & STATE_GROUNDED)    != 0 &&
                   (state & STATE_JUMPING)     == 0 &&
                   (state & STATE_DASHING)     == 0;

    // WALLSLIDING AND NOT GROUNDED
    bool canWallJump = (state & STATE_WALLSLIDING) != 0 &&
                       (state & STATE_GROUNDED)    == 0;
```

Fig. 24. Implementation of (6) and (7) in C# code

The boolean canJump is a direct implementation of (6) in section IV. It uses the bitwise operator & to isolate each the grounded, jumping, and dashing bit to check if it is active (!=0) or inactive (== 0). The boolean canWallJump uses the same operations, being the direct implementation of (7) in section IV.

```
if (Input.GetButtonDown("Jump"))
{
    if (canJump)
    {
        rb.velocity = new Vector2(rb.velocity.x, jumpForce);
        state |= STATE_JUMPING;
        state = (byte)(state & ~STATE_GROUNDED);
    }
    else if (canWallJump)
    {
        rb.velocity = new Vector2(-moveInput * moveSpeed, jumpForce);
        state |= STATE_JUMPING;
        state = (byte)(state & ~STATE_WALLSLIDING);
    }
}
```

Fig. 25. Jump mechanics code

When the player presses the jump button (spacebar), the code runs the jump mechanics, adding velocity upwards. In order to set the states, the program uses the |= bitwise operator in order to activate the intended state and then uses the bitwise AND-NOT to clear the state. When jumping, the grounding state needs to be cleared to satisfy rule 1 (*GJ*), and when wall jumping, wallsliding needs to be cleared to satisfy rule 3 (*JW*).

C. *Dash Logic*

```
void HandleDash()
{
    // NOT DASHING AND NOT HITSTOP
    bool canDash = (state & STATE_DASHING) == 0 &&
                   (state & STATE_HITSTOP) == 0;
```

Fig. 26. Implementation of (8) in C# code

The boolean canDash is a direct implementation of (8) in section IV, utilizing the bitwise operator & and == to check if the state is currently inactive.

```
if (Input.GetKeyDown(KeyCode.LeftShift) && canDash)
{
    state |= STATE_DASHING;
    state = (byte)(state & ~STATE_JUMPING);
    state = (byte)(state & ~STATE_WALLSLIDING);
    dashTimer = dashDuration;
    rb.velocity = new Vector2(moveInput != 0 ? moveInput * dashForce : dashForce, 0);
}

if ((state & STATE_DASHING) != 0)
{
    dashTimer -= Time.deltaTime;
    if (dashTimer <= 0)
        state = (byte)(state & ~STATE_DASHING);
}
```

Fig. 27. Dash mechanics code

If the condition canDash is true then the state will be set to STATE_DASHING using the bitwise operator |= and will clear the jumping state and also clear the wall sliding state to satisfy rule 4 (*DW*). A timer is used to determine the length of the dash.

D. *Wall Slide Logic*

```
void HandleWallSlide()
{
    // WALLSLIDING AND NOT GROUNDED AND NOT DASHING
    bool isWallSliding = isTouchingWall &&
                         (state & STATE_GROUNDED) == 0 &&
                         (state & STATE_DASHING)  == 0 &&
                         (state & STATE_JUMPING)  == 0 &&
                         moveInput != 0;
```

Fig. 28. Implementation of (9) in C# code

The boolean isWallSliding is the direct implementation of (9) in section IV. The bitwise operator & and == is used to verify the inactive states. There are also the touchingWall (*T*) and moveInput (*M*) that is used to validate if the player is touching a wall and moving towards the wall.

```
if (isWallSliding)
{
    state |= STATE_WALLSLIDING;
    rb.velocity = new Vector2(rb.velocity.x,
        Mathf.Max(rb.velocity.y, -wallSlideSpeed));
}
else
{
    state = (byte)(state & ~STATE_WALLSLIDING);
}
```

Fig. 29. Wallsliding mechanics code

The wall sliding mechanic will set the state to currently wall sliding and prevent the player from free-falling by lowering the speed of which the player is falling. When the player has reached the ground (satisfying rule 2), the wall sliding state will be cleared.

E. *Ground Check*

```
void CheckGrounded()
{
    bool grounded = Physics2D.Raycast(transform.position, Vector2.down, 1.1f, groundLayer);
    if (grounded)
    {
        state |= STATE_GROUNDED;
        state = (byte)(state & ~STATE_JUMPING);
        state = (byte)(state & ~STATE_WALLSLIDING);
    }
    else
    {
        state = (byte)(state & ~STATE_GROUNDED);
        if (rb.velocity.y > 0.1f)
        {
            state |= STATE_JUMPING;
        }
        else
        {
            state = (byte)(state & ~STATE_JUMPING);
        }
    }
}
```

Fig. 30. Ground check mechanics code

The ground check uses raycast in order to detect if the player is standing on a ground layer. If they are, then using the bitwise operator |=, the program will set the state to grounded. It will also clear the jumping and wall sliding state to satisfy rule 1 (*GJ*) and rule 2 (*GW*). The not grounded branch clears the grounded state and checks the velocity, setting it to jumping if the player is moving upwards.

F. *Wall Check*

```
void CheckWall()
{
    isTouchingWall = Physics2D.Raycast(transform.position,
        Vector2.right * moveInput, 0.6f, wallLayer);
}
```

Fig. 31. Wall check mechanics code

Similar to the ground check, the wall check also uses Unity's raycast in the direction the player is currently pressing. This prevents wall slide state to be activated when the player is not currently pressing towards a wall.

G. *Hitstop Simulation*

```
if (Input.GetKeyDown(KeyCode.G))
{
    state = STATE_HITSTOP;
    Invoke(nameof(ResetSimulasiHitstop), 0.5f);
}

UpdateHUD();

if ((state & STATE_HITSTOP) != 0) return;
```

Fig. 32. Hitstop input

Pressing G will force the byte to be exactly 16, changing the state to hitstop and clearing every other state.

```
void ResetSimulasiHitstop()
{
    state = (byte)(state & ~STATE_HITSTOP);
    state |= STATE_GROUNDED;
}
```

Fig. 33. Reset hitstop simulation code

After the hitstop timer runs out, the state will be returned to the basic state (grounded).

H. *Bitwise Validation*

```
public bool IsCurrentStateValid()
{
    bool G = (state & STATE_GROUNDED) != 0;
    bool J = (state & STATE_JUMPING) != 0;
    bool D = (state & STATE_DASHING) != 0;
    bool W = (state & STATE_WALLSLIDING) != 0;

    bool isViolationDetected = (G && J) || (G && W) || (J && W) || (D && W);

    return !isViolationDetected;
}
```

Fig. 34. Implementation of (10) as state validation in C# code

isCurrentStateValid is a direct implementation of (10) in section IV. It detects if the current byte is entering a violation state which will be updated in the HUD.

I. *Real Time HUD*

```
void UpdateHUD()
{
    if (stateDisplay == null) return;

    bool isValid = IsCurrentStateValid();
    string validationText = isValid
        ? "<color=green>VALID (Aman)</color>"
        : "<color=red>CONTRADICTION (Error!)</color>";

    stateDisplay.text =
        $"STATE BITS: {System.Convert.ToString(state, 2).PadLeft(8, '0')}\n" +
        $"GROUNDED      [{((state & STATE_GROUNDED)     != 0 ? "1" : "0")}]\n" +
        $"JUMPING       [{((state & STATE_JUMPING)      != 0 ? "1" : "0")}]\n" +
        $"DASHING       [{((state & STATE_DASHING)      != 0 ? "1" : "0")}]\n" +
        $"WALLSLIDING   [{((state & STATE_WALLSLIDING)  != 0 ? "1" : "0")}]\n" +
        $"HITSTOP       [{((state & STATE_HITSTOP)      != 0 ? "1" : "0")}]\n" +
        $"SYSTEM STATUS: {validationText}";
}
```

Fig. 35. HUD code

In order to validate the states, an HUD is used to visualize the current bit and also send a warning if the player enters an invalid state during runtime.

J. *Runtime Verification*

During runtime, the real time HUD is used to verify the validity of the states. During runtime, it is shown that conflicting states never overlap, with the bitwise masking being easily supervised. The states run as they're supposed to in accordance with the formulated logic in section III and IV.

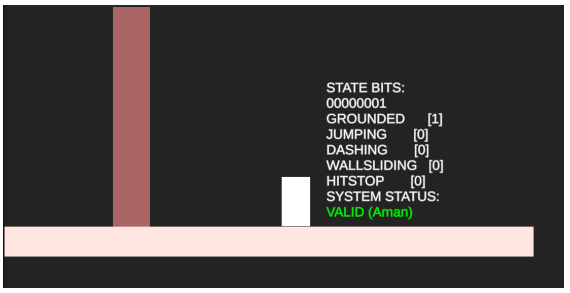

Fig. 36. Grounded state

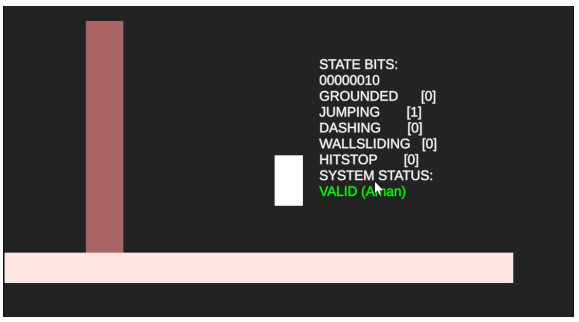

Fig. 37. Jumping state

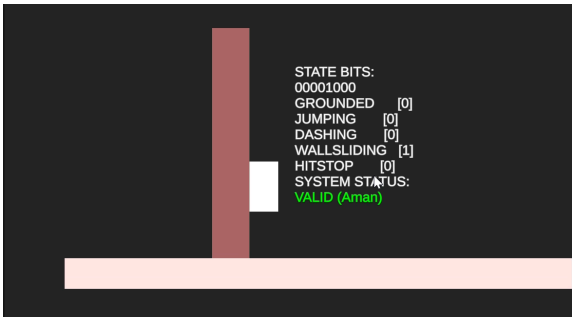

Fig. 38. Wall slide state

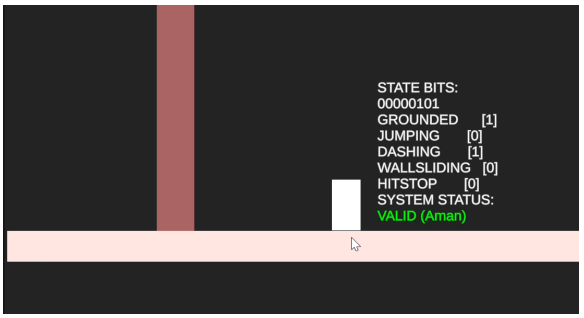

Fig. 39. Dashing state

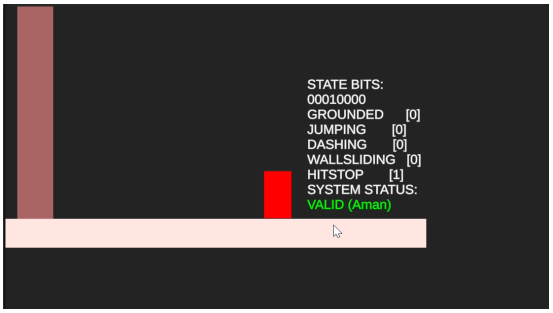

Fig. 40. Hitstop state

VI. Conclusion

By using Boolean algebra and bitwise masking in player state management, the program proves to be effective in creating a system that is mathematically verifiable. By encoding 5 different states into a single byte, a validation function that is created with the Karnaugh map method is achieved that can be used to prevent overlapping of different states. Using Boolean algebra to verify the states has proven to be a mathematically correct approach in preventing bugs and logic errors. This approach can be used as an alternative to the more intuition-based nested if-else method, providing a more accurate and formally provable method.

Video Link at Youtube

https://youtu.be/_lzWnCPMvBc

Repository Link

https://github.com/unadd-rn/Matdis---13525122

Acknowledgment

I would like to express my sincere gratitude to my lecturer, Pak Rinaldi Munir, for his valuable guidance and teaching throughout the semester. Without his support I wouldn't be able to bring this work to completion. I also wish to thank my parents and my sister for their continuous encouragement and support during my studies and the writing process. And lastly, I express my gratitude to the numerous resources that allowed me to deepen my understanding in the subject. This paper would not have been possible without them.

References

[1] Newzoo. "Global Games Market Report Q2 2026 Update." Newzoo Market Research. https://newzoo.com/resources/blog/global-games-market-q2-2026 (accessed June 18, 2026).

[2] Finite State Machines. *Brilliant.org*. https://brilliant.org/wiki/finite-state-machines/ (accessed June 16, 2026)

[3] Munir, Rinaldi. 2020. "Aljabar Boolean (Bagian 1)". https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Aljabar-Boolean-(2020)-bagian1.pdf (accessed on June 18, 2026) .

[4] GeeksforGeeks. "Boolean Search." GeeksforGeeks. https://www.geeksforgeeks.org/dsa/boolean-search/ (accessed June 19, 2026).

[5] Microsoft. 2026. Bitwise and shift operators (C# reference). Microsoft Learn. https://learn.microsoft.com/en-us/dotnet/csharp/languagereference/operators/bitwise-and-shift-operators (accessed on June 19, 2026)

[6] Munir, Rinaldi. 2020. "Aljabar Boolean (Bagian 2)". https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Aljabar-Boolean-(2020)-bagian2.pdf (accessed on June 18, 2026) .

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 1 Juni 2025

Nadia Aulia Syafarani 13525122