**<Hexle: unlimited>**

**Team Number**: 53

*\*Instructions*:

1. ***Directly*** *work on your report in this Google doc. Do not work elsewhere and paste it here.*
2. *Use the template formatting, if it is a heading (e.g., Heading 1, Heading 2 etc.), format it as such so the* ***Table of Contents*** *will reflect it.*
3. *The point of this Google doc is to be a* ***live*** *document that you continually update as you work on your 1D project. The teaching staff will be taking a look at these documents as you work on them to see how your team is progressing.*

Adelaine Ruth Hanako Suhendro - 1007059

Avitra Phon - 1006946

Muvil Kenal Kothari - 1006885

Khoo Yong Xuan - 1006962

Raphael Xujie Yip - 1006657

Muhammad Asyraf Bin Omar - 1006938

Table of Contents

[**Introduction** 3](#_Toc162733311)

[**Game Design** 4](#_Toc162733312)

[*Description of the Game* 4](#_Toc162733313)

[*User Manual* 4](#_Toc162733314)

[*Game Demonstration* 5](#_Toc162733315)

[*Design Inspirations* 6](#_Toc162733316)

[**Electronic Design** 7](#_Toc162733317)

[*Input/Output* 7](#_Toc162733318)

[*Datapath* 10](#_Toc162733319)

[*Regfile* 11](#_Toc162733320)

[*Control Signals* 12](#_Toc162733321)

[*FSM* 13](#_Toc162733322)

[**Budget** 23](#_Toc162733323)

[**2D** 24](#_Toc162733324)

[Objective 1: Streamlining Efficiency and Costs 24](#_Toc162733325)

[*Objective 2: Enhancing User Engagement* 24](#_Toc162733326)

[*Objective 3: Upholding Sustainability, Diversity, and Inclusion* 24](#_Toc162733327)

[**Sustainability and Inclusivity** 25](#_Toc162733328)

[**Summary** 25](#_Toc162733329)

[**References** 26](#_Toc162733330)

[**Appendix** 27](#_Toc162733331)

[*1.* *ALU Design and Tests* 27](#_Toc162733332)

[*2.* *ALU Additional Modules* 35](#_Toc162733333)

[*3.* *Prototype code + Repo link* 37](#_Toc162733334)

[*4.* *Project Management Log: Team Tasks* 37](#_Toc162733335)

[*5.* *Components’ Specifications* 38](#_Toc162733336)

# **Introduction**

<Describe your idea and prototype briefly, can include a photo of the final prototype here>

Our game draws inspiration from the famous online game Wordle. Straying away from traditional keypad inputs, our prototype features four buttons designed for binary input, where each press corresponds to a binary digit (1 for pressed, 0 for not pressed). These buttons enable players to submit each digit of their guess or delete their attempt as needed.

Upon submission of all four digits, the game provides feedback via a 7-segment display. A green display indicates a correct digit in the correct position, while a yellow display signifies a correct digit in the wrong position.

The figures provided illustrate the prototype of our game model.

Our game design prioritizes minimalism to ensure an intuitive and user-friendly experience. Each component of the prototype was 3D printed, and the necessary hardware was purchased to bring the design to life.

A screenshot of a game

Description automatically generatedA white machine with red buttons and black buttons

Description automatically generatedA white device with red and green buttons

Description automatically generatedIn subsequent sections of the report, we will delve deeper into the intricacies of our game model, including the finite state machine mechanics and datapath diagrams.

Figure 3: Instructions of the game

Figure 2: Exploded view of all components

Figure 1: Final game layout 3D model

# **Game Design**

## Description of the Game

The main goal of Hexle is to correctly guess all 4 hexadecimal digits of a randomly generated 16-bit number, in the least number of attempts. At the start of each game, the user is shown a subtraction operator (-) and a 4-bit number. This number, the hint, is the difference between the *largest digit* and the *smallest digit* of the answer.

After every attempt, the 4 digits entered will either turn green, yellow, or remain white.

|  |  |
| --- | --- |
| Green | Digit exists in the answer and is in the correct position |
| Yellow | Digit exists in the answer but is in the wrong position |
| White | Digit does not exist in the answer |

The player can use this information to refine their guesses, to correctly guess the answer.

## User Manual

When turned on, press any button to initialise the game. Immediately after, the LED panel labelled with the “-“ symbol, will display a hexadecimal number which acts as the hint.

The user will have 4 main interactive buttons they can use:

1. 4 white buttons to key in 0-F (in binary),
2. Black Enter button, and
3. Red delete button.
4. Colour blind switch

For each digit, the user toggles the white buttons to choose their desired digit from (0- F), and presses enter to confirm their choice for that digit. Pressing the red delete button removes the most recent digit entered. Digits are entered in order from left to right and deleted from right to left.

After 4 digits have been entered, the user can press the black Enter to lock in their attempt.

The answer will then be checked if it is valid; for the first guess, the answer must incorporate the hint, meaning the difference between the highest and lowest digit in the answer must correspond to the hint. If the answer is not valid, the user will not be allowed to proceed and must modify their answer before proceeding.

If the answer is valid, it will then be evaluated against the correct answer, and the colours of the digits will change based on the correctness of the guess. After the first attempt, the user is required to put the green digits in its right place and have used all yellow digits in their guess to be considered a valid input. This process continues until the user correctly guesses the answer (all digits = green).

At the end of the game, the correct answer would be displayed with all digits flashed in green. The user can then press Enter/Delete to start another round, with a different number to guess and a new hint will be generated and displayed.

## Game Demonstration

A white rectangular object with buttons and numbers

Description automatically generated

1. Press any button to start.
2. Using the 4 buttons, input your first digit guess where pressing the buttons gives a 1 and leaving it unpressed is a 0. (Eg to input F press all buttons – 1111).
3. Press the black button to enter the first digit.
4. As you press the digit one by one, it will display on the 7-segment unit.
5. Your guess must abide by the max – min hint rule.
6. After keying all 4 digits, press the black button to submit your guess
7. If the guess is a valid guess the counter will increase by 1 and you will move on to the next row to make your second guess
8. If it is an invalid guess, digits won’t change colour. Use the delete button to backspace and re-input your guess.
9. Your digits will turn green if they are in the right place and yellow if they are not.
10. From the second guess onwards, you must follow the correct digits in the green position and must have at least the same number of yellow digits from the previous guess to be a valid input. The hint must still be abided.
11. If the guess is correct, all digits will be green, and user can press the enter/delete button to start a new game.

## Design Inspirations

<Illustrate where you got your design ideas from, and how you have improved on it>

This puzzle game was initially inspired by [Wordle](https://www.nytimes.com/games/wordle/index.html). While Hexle shares some similarities with Wordle, it takes a distinct approach in the game mode variation.

* + - 1. Hexle offers greater depth by using hexadecimal digits (0-F) instead of letters, offering a wider range of possibilities and a steeper learning curve compared to the five-letter words in Wordle. This caters to a more niche audience who enjoys a complex numeric challenge.
      2. Unlike Wordle, which relies heavily on vocabulary knowledge, Hexle emphasizes logical deduction. Players need to analyse the color-coded feedback to decipher the placement and existence of hexadecimal digits. This is further reinforced by the inclusion of the hint feature and forcing guesses to abide by the hint to be considered valid.

1. Hexle also doesn't limit players to just one guess per day. Users can keep playing and guessing new randomly generated numbers, making it more replayable than the daily format of Wordle.
2. In addition to the hint, players are also expected to use all the correctly placed green digits and at least all the yellow digits in their subsequent guess, guiding their guesses to have logic rather than just throwing lucky numbers.
3. To add on to the level of difficulty, digit inputs are done in binary instead of keypad presses

In essence, Hexle borrows the engaging concept of guessing an answer with color-coded feedback but injects a fresh layer of complexity by utilizing hexadecimal digits and the inclusion of hints. This shift makes the game more suitable for those who enjoy logic puzzles and mathematical challenges.

A screenshot of a game

Description automatically generatedA screenshot of a game

Description automatically generated

# **Electronic Design**

*Lots of diagrams are expected here.*

## Input/Output signals

|  |  |  |
| --- | --- | --- |
| **Input** | **Expected Values** | **Description** |
| 4 White latching switches | Format: c{digit[3], digit[2], digit[1], digit[0]}  Digit{n} refers to the nth switch, with digit[0] being the LSB and digit[3] being the MSB | 0: 0000  1:0001 2: 0010  3: 0011  4: 0100  5:0101  6: 0110 7: 0111 8: 1000  9: 1001  A: 1010  B:1011 C: 1100 D: 1101 E:1110 F:1111 |
| High Contrast latching button | 1/0 |  |
| Select Button (Black momentary switch) | 1/0 | Confirm digit/answer  Reset Game when game ends |
| Delete Button (Red momentary switch) | 1/0 | Delete digit  Reset Game when game ends |

|  |  |  |
| --- | --- | --- |
| **Output** | **Expected Values** | **Description** |
| 7 bit (segment) for 7 segment data | Digits: 0 to F | Segment selection: 7-seg module will split data into 4x4 bits, translating data (0x0 to 0xF) into relevant display for each digit, each digit will then be translated into 7 bits based on its 7-segment display.  (Format: segment A-G on the 7segment, segment A is the LSB, segment G is the MSB. DP is ignored as it is not used.)  How to Set up Seven Segment Displays on the Arduino - Circuit Basics  0: 0111111  1: 0000110  2: 1011011  3: 1001111  4: 1100110  5: 1101101  6: 1111101  7: 0000111  8: 1111111  9: 1100111  A: 1101111  B: 1111100  C: 0111001  D: 1011110  E: 1111001  F: 1110001  RGB input: 7-seg module will split data into 2x4 bits, translating data into relevant colours for each digit.  In order of priority, for each 4-bit number:  If index 2 = 1 -> Green  Else if Index 1 = 1 -> Yellow  Else if Index 0 = 1 -> White  Else Off |
| 3 Bit (RGB) for RGB selection | Digits:  000 (off)  100 (red)  010 (green)  001 (blue)  110 (yellow)  011 (cyan)  101 (magenta) | Chooses colour to display on the 7 segment unit |
| 4 Bit (digit) for selection of 7 segment unit. | Digits 0000- 1000, 1100, 1101  (since 11 7-segment units are used) | Chooses 7 segment unit to send RGB data to out of the 11 units in operation. |

## UI Display Output

|  |  |  |
| --- | --- | --- |
| Output | Expected value | Description |
| Current attempt | 0x0000 to 0xFFFFF | 4-bit x 4 digits |
| Previous attempt | 0x0000 to 0xFFFFF | 4-bit x 4 digits |
| Hint | 0 to F | Max – Min |
| Counter | 0 to 99 |  |

## Datapath

<Describe the datapath of your game, how many regs used, addressing, control signals, etc>

[Draw.io Datapath file](https://drive.google.com/file/d/1INxOC79JWzA0HZXAkk2mxj6VIw9Il7S8/view?usp=sharing)A diagram of a computer

Description automatically generated

### Regfile

Key Statistics:

* 12 16-bit registers (4-bit addressable)
* 2 addressable combinational READ ports
* 15 fixed combinational READ ports
* 1 sequential WRITE port

|  |  |  |  |
| --- | --- | --- | --- |
| **Address (Total: 20)** | **Description** | **Format** | **Function** |
| 0x0 | Correct Answer | Expected values: 0x0000 to 0xFFFF | Answer to evaluate the user’s attempts. Randomly generated by FPGA inbuilt RNG module |
| 0x1 to 0x2 (2) | Attempts 1 and 2 | Expected values: 0x0000 to 0xFFFF | Provides 8x 7-seg data for which digits to always display, 16-bit will be translated by 7-seg module into valid input |
| 0x3 to 0x4 (2) | RGB Data for Attempts 1 and 2 | Expected values for 1 digit (0-F 4 bit):  0x0007: Green  0x0003: Yellow  0x0001: Red  0x0000: Off  Expected values for 1 attempt after computation (4 0-F digits, 16 bit):  Any n=4 permutation of the 3 options above (e.g. 0xFAF0) | Provides 8x 7-seg data for which colours to always display, 16-bit will be translated by 7-seg module into valid input (Refer to Input/Output section) |
| 0x5 to 0x6 (2) | Counter RGB  Counter Digit |  | Displays the number of attempts |
| 0x7 to 0x8 (2) | Hint Operator RGB and Digit Register | Sub  8 LSB: 0x0 to 0xE1 | Display the hint for the specific round |
| 0xA to 0xF | Temp Register |  |  |

### Control Signals

|  |  |  |
| --- | --- | --- |
| **Control Signals** | **Description** | **Expected Values** |
| alufn (6 bit) | ALUFN signal | Refer to ALU section in Appendix |
| we (1 bit) | Write enable | 0: Writing disabled  1: Writing enabled |
| ra (4 bit) | Register A Address (read) | 0x0000 to 0x0013 |
| rb (4 bit) | Register B Address (read), can also be read as a literal when bsel = 001 |
| rc (4 bit) | Register C Address (write) |
| bsel (3 bit) | Used to select whether to use rb as an address, to read it as a literal, or to access 3 other constants used in computation. | 000: Rb\_data  001: rb  010: 0x0010  011: 0x0100  100: 0x1000 |
| wdsel (1 bit) | Used for resetting registers at the end of the game, else is transparent | 0: ALU output  1: 0x0000 |
| Ra\_data | Bootstrapped connection to FSM for branching |  |

## 

## FSM

A screenshot of a computer

Description automatically generated[Draw.io FSM file](https://drive.google.com/file/d/1INxOC79JWzA0HZXAkk2mxj6VIw9Il7S8/view?usp=sharing)

**REGFILE Addresses:**

0x0: Correct Answer Reg

0x1 to 0x2 (2): Digit Data Regs

0x3 to 0x4 (2): RGB Data Regs

0x5: RGB Counter

0x6: Counter Reg

0x7: RGB Hint

0x8: Hint Reg

0xA to 0xF: Temp Regs to store digit values

|  |  |
| --- | --- |
| **State 1** | |
| **Initialization** | |
| Display | NIL |
| 7 Seg | NIL |
| RGB | NIL |
| **Answer Generation** | **Using a random number generator, a 16-bit answer is created and stored to a register** |
| Display | NIL |
| Storage Register | 0x0 |
| **Max Digit** | **Using the MAX ALUFN, store the highest digit in a temporary register** |
| Display | NIL |
| Input A | 0x0 |
| Input B | NIL |
| Storage Register | 0xE |
| ALUFN | MAX   * Takes in a 16-bit value and returns the largest 4-bit value |
| **Min Digit** | **Using the MIN ALUFN, store the smallest digit in a temporary register** |
| Display | NIL |
| Input A | 0x0 |
| Input B | NIL |
| Storage Register | 0xF |
| ALUFN | MIN   * Takes in a 16-bit value and returns the smallest 4-bit value |
| **Hint Calculation** | **Takes the largest and smallest 4-bit values stored in the temporary registers and subtracts them** |
| Display | 7 Seg shows the value of largest - smallest |
| Input A | 0xE (Contains the max value) |
| Input B | 0xF (Contains the min value) |
| Storage Register | 0xD |
| ALUFN | SUB   * Take 2 inputs and subtracts them |

|  |  |
| --- | --- |
| **State 2 A** | |
| **Attempt 1 Digit 1** | **When the first digit is pressed in binary and enter is pressed** |
| **a.** | **Value from 4 buttons is decoded and stored in a temporary register** |
| Display | NIL |
| Storage Register | 0xF |
| **b.** | **First digit is shifted to the left by 12 spaces to put it in the leftmost digit space [0x000W -> 0xW000]** |
| Display | NIL |
| Input A | 0xF (where the input value is stored) |
| Input B | 1100 (absolute value 12) |
| Storage Register | 0xA |
| ALUFN | SHL   * Takes in 2 inputs and shifts input a by b spaces to the left |
| **c.** | **Add the value of 0xF to 0x1 which is the register to store the first attempt value** |
| Display | NIL |
| Input A | 0x1 (register to store attempt 1 value) |
| Input B | 0xF (contains the value of the first digit shifted by 12 spaces) |
| Storage Register | 0x1 |
| ALUFN | ADD   * Adds the 16-bit value in 0x1 with 0xF [0x0000 + 0xW000 = 0xW000] |
| **d.** | **Change the colour of the first 7 segment to white to display the first digit** |
| Display | First input digit in white |
| Input A | 0x3 (register to store the RGB data for the first attempt) |
| Input B | 0x1000 (use BSEL 100 to give pre-coded absolute value) |
| Storage Register | 0x3 |
| ALFUN | ADD   * Add the 16-bit RGB data of attempt 1 with 0x1000 to give a value of 1 in the 4 MSB bit since white is coded to turn on when value is 1 [0x0000 + 0x1000 = 0x1000] |
| **e.** | **Assuming the button delete is pressed, remove the latest digit** |
| Display | First input digit in white |
| Input A | 0x1 (register to store attempt 1 value) |
| Input B | 0xA (contains the value of the first digit shifted by 12 spaces) |
| Storage Register | 0x1 |
| ALUFN | SUB   * Subtract the 1st attempt and the stored value of the 1st input digit to delete that specific input [0xW000 – 0xW000 = 0x0000 back to original] |
| **f.** | **Change the colour of the 7 segment to remove the display** |
| Display | NIL |
| Input A | 0x3 (register to store the RGB data for the first attempt) |
| Input B | 0x1000 (use BSEL 100 to give pre-coded absolute value) |
| Storage Register | 0x3 |
| ALFUN | SUB   * Subtract the 16-bit RGB data of attempt 1 with 0x1000 to give a value of 0 in the 4 most significant bit spaces since input digit has been deleted [0x1000 – 0x1000 = 0x0000 no RGB display in the first 7 segment] |

|  |  |
| --- | --- |
| **State 2 B** | |
| **Attempt 1 Digit 2** | **When the buttons digit is pressed for the 2nd time and enter is pressed** |
| **a.** | **Value from 4 buttons is decoded and stored in a temporary register** |
| Display | NIL |
| Storage Register | 0xF |
| **b.** | **Second digit is shifted to the left by 8 spaces to put it in the 2nd digit space [0x000X -> 0x0X00]** |
| Display | NIL |
| Input A | 0xF (where the input digit value is stored) |
| Input B | 1000 (absolute value 8) |
| Storage Register | 0xB |
| ALUFN | SHL   * Takes in 2 inputs and shifts input a by b spaces to the left |
| **c.** | **Add the value of 0xF to 0x1 which is the register to store the first attempt value** |
| Display | NIL |
| Input A | 0x1 (register to store attempt 1 value) |
| Input B | 0xF (contains the value of the second digit shifted by 8 spaces) |
| Storage Register | 0x1 |
| ALUFN | ADD   * Adds the 16-bit value in 0x1 with 0xF [0xW000 + 0x0X00 = 0xWX00] |
| **d.** | **Change the colour of the second 7 segment to white to display the second digit** |
| Display | First and Second input digit in white |
| Input A | 0x3 (register to store the RGB data for the first attempt) |
| Input B | 0x0100 (use BSEL 011 to give pre-coded absolute value) |
| Storage Register | 0x3 |
| ALFUN | ADD   * Add the 16-bit RGB data of attempt 1 with 0x0100 to give a value of 1 in the 2nd digit space bit since white is coded to turn on when value is 1 [0x1000 + 0x0100 = 0x1100] |
| **e.** | **Assuming the button delete is pressed, remove the latest digit** |
| Display | First and Second input digit in white |
| Input A | 0x1 (register to store attempt 1 value) |
| Input B | 0xB (contains the value of the second digit shifted by 8 spaces) |
| Storage Register | 0x1 |
| ALUFN | SUB   * Subtract the 1st attempt and the stored value of the 2nd input digit to delete that specific input [0xWX00 – 0x0X00 = 0xW000 back to 1st input] |
| **f.** | **Change the colour of the 7 segment to remove the display** |
| Display | First input digit in white |
| Input A | 0x3 (register to store the RGB data for the first attempt) |
| Input B | 0x0100 (use BSEL 011 to give pre-coded absolute value) |
| Storage Register | 0x3 |
| ALFUN | SUB   * Subtract the 16-bit RGB data of attempt 1 with 0x0100 to give a value of 0 in the 2nd digit space since input digit has been deleted [0x1100 – 0x0100 = 0x1000 no RGB display in the second 7 segment] |

|  |  |
| --- | --- |
| **State 2 C** | |
| **Attempt 1 Digit 3** | **When the buttons digit is pressed for the 3rd time and enter is pressed** |
| **a.** | **Value from 4 buttons is decoded and stored in a temporary register** |
| Display | NIL |
| Storage Register | 0xF |
| **b.** | **Third digit is shifted to the left by 4 spaces to put it in the 3rd digit space [0x000Y -> 0x00Y0]** |
| Display | NIL |
| Input A | 0xF (where the input digit value is stored) |
| Input B | 0100 (absolute value 4) |
| Storage Register | 0xC |
| ALUFN | SHL   * Takes in 2 inputs and shifts input a by b spaces to the left |
| **c.** | **Add the value of 0xF to 0x1 which is the register to store the first attempt value** |
| Display | NIL |
| Input A | 0x1 (register to store attempt 1 value) |
| Input B | 0xF (contains the value of the third digit shifted by 4 spaces) |
| Storage Register | 0x1 |
| ALUFN | ADD   * Adds the 16-bit value in 0x1 with 0xF [0xWX00 + 0x00Y0 = 0xWXY0] |
| **d.** | **Change the colour of the third 7 segment to white to display the third digit** |
| Display | First and Second and Third input digit in white |
| Input A | 0x3 (register to store the RGB data for the first attempt) |
| Input B | 0x0010 (use BSEL 010 to give pre-coded absolute value) |
| Storage Register | 0x3 |
| ALFUN | ADD   * Add the 16-bit RGB data of attempt 1 with 0x0010 to give a value of 1 in the 3rd digit space bit since white is coded to turn on when value is 1 [0x1100 + 0x0010 = 0x1110] |
| **e.** | **Assuming the button delete is pressed, remove the latest digit** |
| Display | First and Second and Third input digit in white |
| Input A | 0x1 (register to store attempt 1 value) |
| Input B | 0xC (contains the value of the third digit shifted by 4 spaces) |
| Storage Register | 0x1 |
| ALUFN | SUB   * Subtract the 1st attempt and the stored value of the 3rd input digit to delete that specific input [0xWXY0 – 0x00Y0 = 0xWX00 back to 2nd input] |
| **f.** | **Change the colour of the 7 segment to remove the display** |
| Display | First and Second input digit in white |
| Input A | 0x3 (register to store the RGB data for the first attempt) |
| Input B | 0x0010 (use BSEL 010 to give pre-coded absolute value) |
| Storage Register | 0x3 |
| ALFUN | SUB   * Subtract the 16-bit RGB data of attempt 1 with 0x0010 to give a value of 0 in the 3rd digit space since input digit has been deleted [0x1110 – 0x0010 = 0x1100 no RGB display in the third 7 segment] |

|  |  |
| --- | --- |
| **State 2 D** | |
| **Attempt 1 Digit 4** | **When the buttons digit is pressed for the 4th time and enter is pressed** |
| **a.** | **Value from 4 buttons is decoded and stored in a temporary register** |
| Display | NIL |
| Storage Register | 0xF |
| **b.** | **Fourth digit is not shifted as it is in its place already** |
| Display | NIL |
| Storage Register | 0xD |
| **c.** | **Add the value of 0xF to 0x1 which is the register to store the first attempt value** |
| Display | NIL |
| Input A | 0x1 (register to store attempt 1 value) |
| Input B | 0xD (contains the value of the fourth digit) |
| Storage Register | 0x1 |
| ALUFN | ADD   * Adds the 16-bit value in 0x1 with 0xF [0xWXY0 + 0x000Z = 0xWXYZ] |
| **d.** | **Change the colour of the fourth 7 segment to white to display the third digit** |
| Display | First and Second and Third and Fourth input digit in white |
| Input A | 0x3 (register to store the RGB data for the first attempt) |
| Input B | 0x1 (absolute value 1) |
| Storage Register | 0x3 |
| ALFUN | ADD   * Add the 16-bit RGB data of attempt 1 with 0x1 to give a value of 1 in the 4th digit space bit since white is coded to turn on when value is 1 [0x1110 + 0x0001 = 0x1111] |
| **e.** | **Assuming the button delete is pressed, remove the latest digit** |
| Display | First and Second and Third and Fourth input digit in white |
| Input A | 0x1 (register to store attempt 1 value) |
| Input B | 0xD (contains the value of the fourth digit) |
| Storage Register | 0x1 |
| ALUFN | SUB   * Subtract the 1st attempt and the stored value of the 4th input digit to delete that specific input [0xWXYZ – 0x000Z = 0xWXY0 back to 3rd input] |
| **f.** | **Change the colour of the 7 segment to remove the display** |
| Display | First and Second and Third input digit in white |
| Input A | 0x3 (register to store the RGB data for the first attempt) |
| Input B | 0x1 (absolute value 1) |
| Storage Register | 0x3 |
| ALFUN | SUB   * Subtract the 16-bit RGB data of attempt 1 with 0x1 to give a value of 0 in the 4th digit space since input digit has been deleted [0x1111 – 0x0001 = 0x1110 no RGB display in the fourth 7 segment] |

|  |  |
| --- | --- |
| **State 2 E** | |
| **Check answer requirements** | **Check whether attempt 1 satisfies the hint requirement** |
| **A1 Max Digit** | **Using the MAX ALUFN, store highest digit of attempt 1 in a temporary register** |
| Display | All 4 digits on the 7 segments in white |
| Input A | 0x1 |
| Input B | NIL |
| Storage Register | 0xF |
| ALUFN | MAX   * Takes in a 16-bit value and returns the largest 4-bit value |
| **A1 Min Digit** | **Using the MIN ALUFN, store the smallest digit in a temporary register** |
| Display | All 4 digits on the 7 segments in white |
| Input A | 0x1 |
| Input B | NIL |
| Storage Register | 0xE |
| ALUFN | MIN   * Takes in a 16-bit value and returns the smallest 4-bit value |
| **A1 Hint Calculation** | **Takes the largest and smallest 4-bit values stored in the temporary registers and subtracts them** |
| Display | All 4 digits on the 7 segments in white |
| Input A | 0xE (Contains the max value) |
| Input B | 0xF (Contains the min value) |
| Storage Register | 0xE |
| ALUFN | SUB   * Take 2 inputs and subtracts them |
| **A1 Check Hint** | **Compares hints in 0xD and 0xE to see if they match and requirements are met** |
| Display | All 4 digits on the 7 segments in white |
| Input A | 0xE |
| Input B | 0xD |
| Storage Register |  |
| ALUFN | CMPEQ   * Take in the hint of attempt and compares if its equals to attempt 2 |
| **Branch** | **If output of CMPEQ is 1 and hints match, continue to check attempt 1 answer else go back to A1D4 and edit attempt** |
| **Counter** | **Add the counter by 1** |
| Display | All 4 digits on the 7 segments in white |
| Input A | 0x6 |
| Input B | 1 (absolute value) |
| Storage Register | 0x6 |
| ALUFN | ADD   * Add 1 to the counter attempt |
| **Counter RGB** | **Displays a white colour for the counter 7 segment** |
| Display | All 4 digits on the 7 segments in white and the counter 7 segment |
| Input A | 0x5 |
| Input B | 1 (absolute value) |
| Storage Register | 0x5 |
| ALUFN | ADD |
| **Check Green** | **Checks if any of the digits match the answer** |
| Display | All 4 digits on the 7 segments in white |
| Input A | 0x0 |
| Input B | 0x1 |
| Storage Register | 0xF |
| ALUFN | GRN   * Matches each 4-bit in the 16-bit input value with the answer to see if it matches and returns 1 |
| **Shift Green** | **Shifts the value of RGB by 2** |
| Display | All 4 digits on the 7 segments in white |
| Input A | 0x0 |
| Input B | 2 (absolute value) |
| Storage Register | 0xF |
| ALUFN | SHL   * Shifts the RGB value by 2 spaces [0x0001 -> 0x0100] since green is in the second index of RGB data for each digit |
| **A1 Merge green** | **Updates the RGB value for attempt 1 and makes digits in the correct place turn green** |
| Display | All 4 digits on the 7 segments in white |
| Input A | 0x0 |
| Input B | 0x3 (Attempt RGB register) |
| Storage Register | 0x3 |
| ALFUN | ADD   * Add the 16-bit RGB data of attempt 1 with 0xF to updates the value In the green position since green is coded to turn on when index 2 is 1 [0x0001 + 0x0100 = 0x0101]. IF all 4 digits RGB is green ie. 0x4444, game ends. |
| **Check Yellow** | **Checks if any of the digits are present in the answer** |
| Display | All 4 digits on the 7 segments in white |
| Input A | 0x0 |
| Input B | 0x1 |
| Storage Register | 0xF |
| ALUFN | YLW   * Matches each 4-bit in the 16-bit input value with the answer to see if it exists and returns 1 |
| **Shift Yellow** | **Shifts the value of RGB by 1** |
| Display | All 4 digits on the 7 segments in white |
| Input A | 0x0 |
| Input B | 1 (absolute value) |
| Storage Register | 0xF |
| ALUFN | SHL   * Shifts the RGB value by 1 space [0x0001 -> 0x0010] since yellow is in the first index of RGB data for each digit |
| **A1 Merge yellow** | **Updates the RGB value for attempt 1 and makes correct digits turn yellow** |
| Display | All 4 digits on the 7 segments in white/green/yellow |
| Input A | 0x0 |
| Input B | 0x3 (Attempt RGB register) |
| Storage Register | 0x3 |
| ALFUN | ADD   * Add the 16-bit RGB data of attempt 1 with 0xF to updates the value In the green position since green is coded to turn on when index 2 is 1 [0x0001 + 0x0010 = 0x0011]. Using another logic, green > yellow > white for every digit displaying the respective colour. |

|  |  |
| --- | --- |
| **State 3** | |
| **Repeat the same steps in State 2 for the second-row attempt** | |
| Branch D1 to get the attempt 1 16-bit RGB input into 4 x 4-bits to check with the attempt 2 | |
| Branch D1 | Branch to check if the digits are abiding by the green and yellow requirements |
| **Extract A1D1** | **Get the attempt 1 16-bit RGB’s first 4-bit** |
| Input A | 0x1 (Attempt 1 register) |
| Input B | 11 (absolute value to get digit 1) |
| Storage Register | 0xE |
| ALUFN | EXT   * Extract the first 4-bits (from the left) from the 16-bit RGB input data |
| **Branch D1 G** | **If the value of RGB of D1 is green [0x0100] do the following** |
| **Extract A2D1** | **Get the attempt 2 16-bit RGB’s first 4-bits** |
| Input A | 0x2 (Attempt register) |
| Input B | 11 (absolute value) |
| Storage Register | 0xF |
| ALUFN | EXT   * Extract the first 4-bits (from the left) from the 16-bit RGB input data |
| **D1G CMPEQ** | **Compare the values of the first digit to see that it matches** |
| Input A | 0xE (Attempt 1 digit 1) |
| Input B | 0xF (Attempt 2 digit 1 RGB) |
| Storage Register | 0xE |
| ALUFN | CMPEQ   * Compare if both the inputs are the same |
| **Branch D1 Y** | **If the value of RGB of D1 is yellow [0x0010] do the following** |
| **Count A1D1 in A1** | **Count the number of times yellow A1D1 exists in attempt 1** |
| Input A | 0xE (Attempt 1 digit 1) |
| Input B | 0x1 (Attempt 1) |
| Storage Register | 0xF |
| ALUFN | CNT   * Loops through the attempt 1 and counts number of times digit 1 appears |
| **Count A1D1 in A2** | **Counts the number of times A1D1 exists in attempt 2** |
| Input A | 0x2 (Attempt 2) |
| Input B | 0xE (Attempt 1 digit 1) |
| Storage Register | 0xE |
| ALUFN | CNT   * Loops through the attempt 2 and counts number of times digit 1 appears |
| **D1Y CMPLE** | **Compare the number of times digit 1 appears in attempt 1 and 2** |
| Input A | 0xF (Number of times digit 1 appears in attempt 1) |
| Input B | 0xE (Number of times digit 1 appears in attempt 2) |
| Storage Register | 0xE |
| ALUFN | CMPLE   * Checks whether the number of times digit 1 appears in attempt 1 is less than or equals to the number of times digit 1 appears in attempt 2 |
| **If the CMPEQ/CMPLE gives an output of 0x1, continue to check digit 2/3/4 else go back to edit your user guess** | |
| **Repeat for the remaining 3 digits using different Rb values to check the conditions for digit 2/3/4** | |
| **Once checked, proceed to check for green/yellow in the new attempt and update the RGB value like in STATE 2E.** | |
| **Migrate A2 (Digit)** | **Move the digits of A2 into A1’s 7-segments** |
| **Migrate A2 (RGB)** | **Move the RGB values of A2 into A1’s 7-segments** |
| **Reset A2** | **Clear data of A2 to allow user to input attempt 3** |

|  |  |
| --- | --- |
| **STATE 4** |  |
| END | Game is over when output of Ra is 0x4444 |
| **Enter/Delete** | **When user then presses enter/delete button, the game resets** |
| Reset A1 | Reset the attempt value of A1 from 0x1 |
| Reset A2 | Reset the attempt value of A2 from 0x2 |
| Reset A1 RGB | Reset the RGB value of A1 from 0x3 |
| Reset A2 RGB | Reset the RGB value of A2 from 0x4 |
| Reset Hint | Reset the hint value from 0x8 |
| Reset Hint RGB | Reset the RGB value of hint from 0x7 |
| Reset Counter | Reset the value of counter from 0x6 |
| Reset Counter RGB | Reset the RGB value of counter from 0x5 |
| Loop to answer generation to start new game | |

# **Budget**

|  |  |  |
| --- | --- | --- |
| **Component** | **Quantity** | **Total Price** |
| 74HC238E Decoder (Texas Instruments) | 28 | $78.40 |
| HOUKEM 1.2-inch RGB 7-Segment Display | 11 | $16.50 |
| Breadboards | 4 (L) + 2(S) + 2(S) | $20 + $6 + $0 |
| Jumper Cables | 120 +100 | $15 + 0 |
| Buttons | 2 (momentary) + 4 (latching)+ 1 (latching) | $6 + $4 + $0 |
| Crocodile Clips | 14 | $7 |
| 74HC04N (inverters) | 2 | $1.80 |
| LED green+ red | 10 + 10 | $0 |
| 10K Ohms Resistors | 22 | $0 |
| Subtotal |  | $154.70 |

\*$0 refers to school provided/ reused from old projects

# **2D**

<Write about efforts you’ve put into your project that aligns with 2D>

Our team has actively addressed the focus areas outlined by YouTwitFace's vision.

## Objective 1: Streamlining Efficiency and Costs

FPGA Optimization: We're actively optimizing the FPGA to handle data processing around the ALU more efficiently. This enhances game speed and resource utilization, aligning with the goal of streamlining development and maintenance.

## Objective 2: Enhancing User Engagement

Upgraded Game Complexity: We've incorporated additional layers of complexity to the game. In our current iteration, the game utilizes the hex subtraction hint and enforces rules that ensure each attempt adheres to the logic, making it more challenging and engaging for users.

Binary Button Design: Users get a challenging touch by having to input their digits in binary using buttons where a press represents 1 while leaving it untouched represents 0 which is intuitive. We’re also prioritizing user comfort by designing buttons with appropriate size and placement.

Good Score Display: We're using 7-segment for our displays which makes a very clear score system ensuring players can easily track their progress with good UI.

Responsive Design: We're ensuring the game reacts intuitively to player actions. This includes aspects like LED and button press response times and counter updates, for a smoother gameplay experience.

## Objective 3: Upholding Sustainability, Diversity, and Inclusion

Use of Recycled Materials: We've utilized recycled materials for the physical components of the prototype like old breadboards, jumper wires and buttons. This reduces our environmental footprint and aligns with sustainable practices.

Component Efficiency: We're striving for a minimalist approach by limiting the number of electronic components used in the game (using only 2 rows of 7 segment displays). This reduces resource consumption and overall environmental impact.

Inclusivity: We're integrating a colourblind mode that replaces the standard game colours with high-contrast alternatives. This involves swapping the green and yellow colours for cyan and magenta, ensuring everyone can distinguish visual cues effectively.

By prioritizing these efforts, we aim to deliver a game that not only fulfills YouTwitFace's vision of an engaging and responsive experience but also reflects values of sustainability and inclusivity.

# **Sustainability and Inclusivity**

<Write about the sustainability and inclusivity aspect of your hardware prototype>

As covered in the earlier section, our project encompasses various practices that reflects the sustainability and inclusivity aspect. In summary,

* + - 1. Use of Recycled Materials for the physical components of the prototype.
      2. Component Efficiency by limiting the number of electronic components used in the game.
      3. Integrating a colourblind mode that replaces the standard game colours with high-contrast alternatives for inclusion of colourblind people.

# **Summary**

<Conclusion, lessons learned, overall experience>

We successfully transformed the OMEGA-16 ALU into a user-friendly and engaging game experience named Hexle. By focusing on user appeal, hardware optimization, sustainable practices, and inclusivity, we believe Hexle has the potential to be a commercially viable product.

Lessons Learned:

In the initial stages, we explored using a VGA interface for displaying the game. However, after careful consideration, we realized that a VGA solution would be overkill for our needs. Since Hexle doesn't require complex graphics, we opted for a more efficient and cost-effective approach using 7-segment displays. It was important to select the right hardware for the project's requirements.

While testing, we also encountered challenges integrating the 7-segment displays due to their differing data sheets and the difference in brightness due to the current requirements. This required rigorous trial-and-error by adding resistors to lower brightness of red etc. This experience shows the importance of thorough research and compatibility testing in hardware development.

Balancing the display update rate with the game's logic and user interaction also required careful consideration of clock cycles. We needed to ensure the displays updated quickly enough to show the correct color without missing a clock cycle, which could lead to flickering or incorrect information.

Overall, the project was a rewarding experience that allowed us to apply our engineering skills to create a fun product. We learnt lessons about selecting the right technology, integrating hardware components, optimizing clock cycles, and the importance of considering the entire user experience. We're confident that Hexle is not just a technically sound game but also one that reflects our learning and application of computation structures.

# **References**

# **Appendix**

## ALU Design and Tests

**1a. Operation Modules**

Found here are the individual operation modules of the FPGA and a brief description on how it is implemented

- **Adder**: Performs addition and subtraction of 16-bit two’s complement (Signed) inputs. Implements **sixteenbit\_rca,** which in turn implements **full\_adder**. Generates 32 bit output, Z (Zero), V (Overflow) and N (Negative). Implemented as suggested in the lab handout.

- **Compare:** Performs comparison operation (CMPEQ, CMPLT, CMPLE) based on alufn signal. Implements **mux\_4** to decide which comparison operation to execute. Implemented as suggested in the lab handout.

- **Boolean:** Performs logical operations (AND, OR, XOR, A (LDR)) based on alufn signal. Implements **mux\_4** to decide which boolean operation to execute. Implemented as suggested in the lab handout.

- **Shifter**: Performs shift operation (SHL, SHR, SRA) based on alufn signal. Implements -

* **x\_bit\_shifter** which in turn implements **mux\_2** to decide which shift operation to execute.
* **x\_bit\_shifter** is able to switch between all of the aforementioned shift operations based on the alufn signal.

- **Multiplier**: Performs multiplication between 16-bit inputs. Implements **rca\_adderarray** which in turn implements **sixteenbit\_rca** which implements **full\_adder**. Implemented as suggested in the lab handout.

- **Rotate:** Performs bit rotation on binary numbers, where bits shifted off one end are reintroduced at the opposite end, preserving all original bits. It implements **x\_bit\_rotator** and handles two operations:

* Rotate Left (ROL): Shifts bits to the left, moving the leftmost bit to the rightmost position.
* Rotate Right (ROR): Shifts bits to the right, moving the rightmost bit to the leftmost position.
* It takes in a 16-bit number to rotate (a), a 4-bit rotate count (b), and a control signal (alufn\_signal) determining the rotation direction.
* Directed by alufn\_signal, the module rotates by the number of positions specified in b, either left or right, ensuring no bit loss by wrapping bits around.
* Utilizes shift registers, multiplexers, and bitwise operations for efficient bit manipulation and rotation.

**1b. Auxiliary Modules**

Found here are supporting modules used for the implementation of the Operation modules.

**- sixteenbit\_rca**: implements 16 **full\_adders** in order to support the **adder** module. Implemented as suggested in the lab handout.

**- full adder**: performs bitwise addition. Implemented as suggested in the lab handout.

**- mux\_4**: chooses which of its 4 inputs is to be executed based on the alufn signals fed into the selector inputs. Supports boolean and compare operations. Implemented as suggested in the lab handout.

**- x\_bit\_shifter**: Implements **mux\_2** units to choose which shift operation to carry out. All shift operations are also coded here instead of in separate modules. Supports **shift** module.

**- mux\_2:** chooses which of its 2 inputs is to be executed based on the alufn signals fed to the selector inputs. Supports **x\_bit\_shifter**. Implemented as suggested in the lab handout.

- **rca\_adderarray**: Implements **sixteenbit\_rca** to create a 15 by 16 full\_adder to be used in the multiplier. Implemented as suggested in the lab handout.

**- x\_bit\_rotator:** Similar to the shifter, implements mux\_2 units to choose which rotate operation to carry out.

**2. Table of Operations**

This is a table showcasing the alufn signal tied to each operation of the FPGA.

|  |  |
| --- | --- |
| **Operator** | **ALUFN** |
| ADD | 000000 |
| SUB | 000001 |
| SHL | 100000 |
| CMPEQ | 110011 |
| CMPLE | 110111 |
| B | 011100 |
| GRN | 000111 |
| YLW | 010111 |
| EXT | 100111 |
| CNT | 110111 |
| MAX | 101111 |
| MIN | 111111 |

**3. Manual Mode**

For the manual mode, users are to input values for A, B, and ALUFN through the IO’s dip switches.This is implemented through a Finite State Machine (FSM) named "Button Controller FSM" is a practical approach. It has 3 main states:

|  |  |  |  |
| --- | --- | --- | --- |
| **Current State** | **Action** | **Next State** | **Notes** |
| State 1 | Press Button (Set A) | State 2 | Set A, ready for B |
| State 2 | Press Button(Set B) | State 3 | Set B, ready for output |
| State 3 | Press Button | State 1 | Output displayed through LEDs |

**4. AutoTesting Mode (Test Cases)**

Test cases are pre-determined, with inputs A and B being pre-assigned, alongside an expected output. This data is stored in a ROM (autotester\_rom).

When AutoTesting Mode is activated, a state machine is used to cycle through the various test cases.

The initial state (Z) starts the test sequence.

Each subsequent state (I to XXXVIII) loads a different test case from a ROM (autotester\_rom) into the ALU inputs (alu\_a, alu\_b, alu\_output, alu\_alufn). The output generated by the ALU is then compared to the answer in the ROM. If the actual answer matches the answer of the ALU, the LED will light up.

The address for the ROM is incremented in each state, effectively iterating through the test cases stored in the ROM.

Once all test cases are exhausted, the state machine returns to the initial state (Z) and restarts the test sequence. Below are features of the implemented FSM:

1. Clock Trigger: The FSM transitions to a new state based on the slow\_clock signal.
2. Address Generation: Based on the current state, the FSM generates an address and sends it to the ROM.
3. ROM Access: The ROM retrieves four data values based on the address.
4. Data Storage and Control: ALU operands are stored in DFFs, and control signals might influence the system's behaviour.
5. ALU Operation: The ALU (potentially) uses stored data for calculations based on the state and control signals.
6. Output Display: The ALU result displays on the LED, while other components might utilize stored data and control signals during processing.

**Top Module (`au\_top`)**

**Functionality**

- The top-level module integrates the ALU with I/O peripherals like LEDs, buttons, and DIP switches on an FPGA board.

- Handles reset synchronization and user input/output through various components.

**Structure**

- Instantiates the `alu` module and interfaces it with physical I/O elements.

- Uses a `reset\_conditioner` to manage reset synchronisation.

- Maps user inputs (DIP switches, buttons) to ALU inputs and displays ALU outputs on LEDs.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Operation** | **State** | **FSM State** | **ALUFN (6-bit)** | **A (16-bit)** | **B (16-bit)** | **out (16-bit)** | **Test name** |
| ADD | Z |  | 000000 | 0000 1100 0010 1011 | 0000 0000 0000 0000 | 0000 1100 0010 1011 | Zero Addition |
| I | 1 | 000000 | 0000 0000 0000 0000 | 0000 1100 0010 1011 | 0000 1100 0010 1011 | Zero Addition |
| II | 2 | 000000 | 0000 1100 0010 1011 | 0001 0011 1101 0111 | 0010 0000 0000 0010 | Postive Numbers |
| III | 3 | 000000 | 0111 1111 1111 1111 | 0111 1111 1111 1111 | 1111 1111 1111 1110 | Boundary Values |
|  |  |  |  |  |  |  |  |
| SUB | IV | 4 | 000001 | 0001 0011 1101 0111 | 0000 0000 0000 0000 | 0001 0011 1101 0111 | Zero Subtraction |
| V | 5 | 000001 | 0000 0000 0000 0000 | 0001 0011 1101 0111 | 1110 1100 0010 1001 | Zero Subtraction |
| VI | 6 | 000001 | 0001 0011 1101 0111 | 0011 1100 0010 1010 | 1101 0111 1010 1101 | Underflow |
| VII | 7 | 000001 | 0000 0000 0000 0000 | 0111 1111 1111 1111 | 1000 0000 0000 0001 | Boundary Values |
|  |  |  |  |  |  |  |  |
| MUL | VIII | 8 | 000010 | 0011 0100 1101 0011 | 0000 0000 0000 0000 | 0000 0000 0000 0000 | Zero Multiplication |
| IX | 9 | 000010 | 0011 0100 1101 0011 | 0000 0000 0000 0001 | 0011 0100 1101 0011 | Multiplication By One |
| X | 10 | 000010 | 0011 0100 1101 0011 | 0101 1001 1010 1101 | 0000 1101 1001 0111 | Postive Numbers |
| XI | 11 | 000010 | 1111 1111 0000 1111 | 1111 0000 0000 1010 | 0000 0110 1001 0110 | Overflow (Optional) |
|  |  |  |  |  |  |  |  |
| AND | XII | 12 | 011000 | 1101 1001 1010 1101 | 0000 0000 0000 0000 | 0000 0000 0000 0000 | All 0s and 1s |
| XIII | 13 | 011000 | 1101 1001 1010 1101 | 1111 1111 1111 1111 | 1101 1001 1010 1101 | All 0s and 1s |
| XIV | 14 | 011000 | 1101 1001 1010 1101 | 1101 1001 1010 1101 | 1101 1001 1010 1101 | Identity Check |
| XV | 15 | 011000 | 1101 1001 1010 1101 | 0010 0110 0101 0010 | 0000 0000 0000 0000 | Complement Check |
|  |  |  |  |  |  |  |  |
| OR | XVI | 16 | 011110 | 1101 1001 1010 1101 | 0000 0000 0000 0000 | 1101 1001 1010 1101 | All 0s and 1s |
| XVII | 17 | 011110 | 1101 1001 1010 1101 | 1111 1111 1111 1111 | 1111 1111 1111 1111 | All 0s and 1s |
| XVIII | 18 | 011110 | 1101 1001 1010 1101 | 1101 1001 1010 1101 | 1101 1001 1010 1101 | Identity Check |
|  |  |  |  |  |  |  |  |
| XOR | XIX | 21 | 010110 | 1101 1001 1010 1101 | 0000 0000 0000 0000 | 1101 1001 1010 1101 | Identity Check |
| XX | 22 | 010110 | 1101 1001 1010 1101 | 1101 1001 1010 1101 | 0000 0000 0000 0000 | Identity Check |
| XXI | 23 | 010110 | 1101 1001 1010 1101 | 0010 01100 0101 0010 | 1111 1111 1111 1111 | Complement Check |
|  |  |  |  |  |  |  |  |
| A | XXII | 24 | 011010 | 1101 1001 1010 1101 | 0000 0000 0000 0000 | 1101 1001 1010 1101 |  |
|  |  |  |  |  |  |  |  |
| SHL | XXIII | 25 | 100000 | 0000 0000 0010 1001 | 0000 0000 0000 0010 | 0000 0000 1010 0100 | Logical Shift |
| XXIV | 26 | 100000 | 0000 0000 0010 1001 | 0000 0000 0000 0000 | 0000 0000 0010 1001 | Zero Shift |
| XXV | 27 | 100000 | 0000 0000 0010 1001 | 0000 0000 0000 1111 | 1000 0000 0000 0000 | Max Shift |
|  |  |  |  |  |  |  |  |
| SHR | XXVI | 28 | 100001 | 0000 0000 0010 1001 | 0000 0000 0000 0010 | 0000 0000 0000 1010 | Logical Shift |
| XXVII | 29 | 100001 | 1000 0000 0000 0000 | 0000 0000 0000 0000 | 1000 0000 0000 0000 | Zero Shift |
| XXVIII | 30 | 100001 | 1000 0000 0000 0000 | 0000 0000 0000 1111 | 0000 0000 0000 0001 | Max Shift |
|  |  |  |  |  |  |  |  |
| SRA | XXIX | 31 | 100011 | 0101 0010 0000 0000 | 0000 0000 0000 0011 | 0000 1010 0100 0000 | Arithmetic Right Shift (postive) |
| XXX | 32 | 100011 | 1101 0010 0000 0000 | 0000 0000 0000 0011 | 1111 1010 0100 0000 | Arithmetic Right Shift (negative) |
|  |  |  |  |  |  |  |  |
| CMPEQ | XXXI | 33 | 110011 | 0000 0000 0000 1010 | 0000 0000 0000 1010 | 0000 0000 0000 0001 | Equality |
| XXXII | 34 | 110011 | 0000 0000 0000 1010 | 0000 0000 0000 1011 | 0000 0000 0000 0000 | Inequality |
|  |  |  |  |  |  |  |  |
| CMPLT | XXXIII | 35 | 110101 | 0000 0000 0000 0100 | 0000 0000 0000 1000 | 0000 0000 0000 0001 | Less Than |
| XXXIV | 36 | 110101 | 0000 0000 0000 0100 | 0000 0000 0000 0010 | 0000 0000 0000 0000 | Greater Than |
| XXXV | 37 | 110101 | 0000 0000 0000 0100 | 0000 0000 0000 0100 | 0000 0000 0000 0000 | Equality |
|  |  |  |  |  |  |  |  |
| CMPLE | XXXVI | 38 | 110111 | 0000 0000 0000 0100 | 0000 0000 0000 1000 | 0000 0000 0000 0001 | Less Than |
| XXXVII | 39 | 110111 | 0000 0000 0000 0100 | 0000 0000 0000 0100 | 0000 0000 0000 0001 | Equality |
| XXXVIII | 40 | 110111 | 0000 0000 0000 0100 | 0000 0000 0000 0010 | 0000 0000 0000 0000 | Greater Than |
|  |  |  |  |  |  |  |  |
| RTL | XXXVI | 41 | 101000 | 1000 0100 0010 0001 | 0000 0000 0000 0000 | 1000 0100 0010 0001 | Zero Rotation |
| XXXVII | 42 | 101000 | 1000 0100 0010 0001 | 0000 0000 0000 1000 | 0010 0001 1000 0100 | Test Rotation |
|  |  |  |  |  |  |  |  |
| RTR | XXXVIII | 43 | 101100 | 0001 0010 0100 1000 | 0000 0000 0000 0000 | 0001 0010 0100 1000 | Zero Rotation |
| XXXVI | 44 | 101100 | 0001 0010 0100 1000 | 0000 0000 0000 1000 | 0100 1000 0001 0010 | Test Rotation |

## ALU Additional Modules

1. GREEN

Inputs: a[16], b[16]

Outputs: out[16]

Compares 4-bit sequences of a and b at matching indexes, returning 0x1 if they are bitwise equal and 0x0 otherwise (strictly 3:0, 7:4, 11:8, 15:12). Concatenate result into 1 16-bit output.

Implementation: cmp compare[4] (CMPEQ) OR Lucid inequalities

1. YELLOW

Inputs: a[16], b[16]

Outputs: out[16]

For each 4-bit sequence of b (3:0/7:4/11:8/15:12), checks if it matches a[3:0], a[7:4], a[11:8], or a[15:12], returning 0x1 if there is >1 match and 0x0 otherwise. Concatenate the 4 results into 1 16-bit output.

Implementation: cmp compare[4][4] (CMPEQ), result = | compare.q[0][3:0] (for the last digit) OR Lucid inequalities

1. MAX/MIN

Inputs: a[16]

Output: out[16]

MAX: Returns the largest 4-bit sequence of a (3:0/7:4/11:8/15:12).

MIN: Returns the smallest 4-bit sequence of a (3:0/7:4/11:8/15:12).

Implementation: cmp compare (CMPLT) OR Lucid inequalities , for loop

1. CNT

Inputs: a[16], b[4]

Output: count[4]

Implementation: Counts number of times 4 bit b exists in 16 bit input a using == operator for each 4 bits in 16 bit input a. Returns sum of True instances.

1. EXT

Inputs: a[16], b[2]

Output: out[4]

Implementation: Extracts 4 bit output from the 16 bit input based on the 2 bit selector.

## Hardware

The hardware is made of 3 main components, the 7-segment decoders, the unit decoders, and the inverters.

7-Segment decoder

Each 7 segment has 7 segments in use and since 11 units are in operation, there needs to be a connection to all 77 pins. Hence, we use a 7:128 decoder system made of 19 3:8 decoders. Each coder takes in up to 3 inputs, a Write Enable Signal and has up to 8 outputs. The system takes in a 7-bit input and outputs a signal to a single pin. The system is further broken down to 3 layers.

Layer 1 consist of 1 decoder. It takes in the MSB of the 7 bits as its sole input. Its Write enable is constantly on. It has 2 outputs wired to the Write enable signals of the decoders on layer 2.

Layer 2 consist of 2 decoders. They both take in the next 3 most significant bit as its inputs. Their Write enable are connected to the outputs of layer 1 and they each have 8 outputs, connected to the 16 write enable signals of layer 3.

Layer 3 consist of 16 decoders, each taking in the 3 LSB as its inputs. Their Write enable is connected to the outputs of layer 2. The outputs of the decoders in this layer are connected to the pins of the 7 segments. Hence a total of 77 outputs are used.

The system takes a 1 7bit signal from the FPGA every clock cycle and activates a particular segment. When done fast enough, the FPGA cycles between its 7 bits fast enough to display a full digit on the 7 segment.

Unit Decoders

The unit decoders chooses a 7 segment unit and activates a particular RGB signal.

It takes in 4bit digits input and a 3bit RGB input. It outputs RGB signals for all 11 7 segment units, totalling to 33 outputs.  
Since there are 3 RGB signal (one each for Red Green and Blue) and 11 units, there are 3 4:16 decoder set-ups. The decoder set-up is similar to the 7-segment decoders, only having the first 2 layers. However, instead of having the write enable of the first layer be constantly on, they are instead connected to the RGB input from the FPGA (each of the RGB bit is connected to the write enable of the respective 4:16 set-up). Thus, only when there is a positive signal for that particular RGB bit will there be a signal written to the RGB of the chosen unit. Since there are 3 4:16 set-up at the same time, each 7-segment unit is able to receive a red, green and blue signal simultaneously, allowing for a combination of colours.

Inverters

The 7-segments used are common anode, meaning to power RGB, 0 is needed instead of the usual 1. Thus the outputs of the unit decoders are run through the inverters to invert the signal to correctly display the desired colour for each 7-segment unit. Furthermore, resistors (10k) are added to the outputs of these inverters to reduce the voltage through the 7-segments. This is crucial to balance out the colours as red was overpowering green and blue. What this meant was that if red and green were enabled at the same time, instead of the desired yellow output, the 7segment will instead appear red as it “overpowers” green to a large extent. Hence, the resistors allows the colours to be more balanced out, allowing for them to mix and combine.

## Prototype code + Repo link

<https://github.com/50002-computation-structures/1d-project-group-53.git>

<https://github.com/muvilk/Hexle>

## Project Management Log: Team Tasks

Yong Xuan: Work on the decoders and test it. Test all the electronic parts also.

Raphael: Set up the FGPA software, write all the ALUFN modules and create the FSM/Datapath.

Muvil: Work on the control signals and report documentation. Put the physical prototype into a video.

Angie: CAD the 3D model of the physical prototype and work on building the game box.

Asyraf: Clean up the FPGA code and help with the AU top construction. Think of the input and outputs of each electronic piece.

Adelaine: Work on the poster and do the wiring of the physical prototype. Ensure rigidness and conduct stress test.

Software:

* ALU Module Design
  + MIN
  + MAX
  + GRN
  + YLW
  + MOD
* 7-Seg Module Update (RGB and Digits)
* REGFILE Creation
* FSM
* Keypad input module
* Figure out multiplex system

Hardware:

* Digit Decoder (30\*7 = 210) therefore 8 bit signal
* RGB Decoder (30\*3 = 90) therefore 7 bit signal
* Keypad Plan and Test
* RGB 7-Seg Plan and Test

New Version:

* 14\*7 = 98 (7 bit signal)
* 14\*3 = 42 (6 bit signal)

## Components’ Specifications

* + - 1. 74HC238 3 to 8 Decoder: [Decoder / Demultiplexer, HC Family, 1 Gate, 3 Input, 8 Output, 5.2 mA, 2 V to 6 V, SOIC-16](https://sg.element14.com/nexperia/74hc138d/ic-74hc-cmos-smd-74hc138-soic16/dp/1201320?st=decoder3%20input%208%20output)
      2. 1.2 inch RGB 7-Seg: <https://www.alibaba.com/product-detail/HOUKEM-12011-ARGB-Tri-colors-1_1600231385963.html>
      3. Red Button: <https://sg.element14.com/apem/1415nc-red/switch-spdt-3a-12vdc-solder-red/dp/1082276?st=red%20button>
      4. Green Button: <https://sg.element14.com/multicomp/r13-23b-05-bg/switch-spst-latching-green/dp/1634636?st=green%20button>