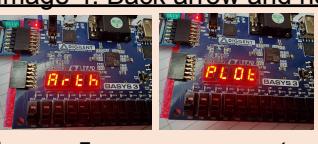
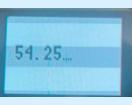
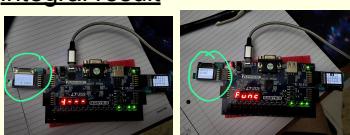
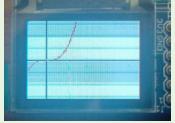


## Personal and Team Improvements

Student and Improvement Name	Improvement Description	Images / Photos
Team "S2_07" "Smart Graphing Calculator"	<p><b>SYStor</b> performs <b>arithmetic operations</b> (+, -, *, /) with decimal support using Q16.16 representation. It can also <b>plot functions</b> of up to 3rd order polynomials.</p> <p>The rendered graph can be panned in all directions and zoomed for detailed viewing beyond the default screen boundaries.</p> <p>Users can view the <b>table of y values</b> for each specific x-value of the function, starting from a user-selected x.</p> <p>An integral mode takes in the desired bounds from the user to find the <b>area under the graph</b>. Once entered, the result is provided, and the area under the curve within the specified bounds is also <b>shaded</b>.</p>	
Student A: Wayne Choy "Menus and Seven Segment Display"	<ol style="list-style-type: none"> <li>1. The “main menu” displays the calculator’s name, “SYStor”, along with two buttons: “Start” and “Help”. The users can use the button “BTNC” to confirm their selection. And use “BTNU” and “BTND” to navigate. There is also a cursor to tell the user their current selection. (image 1)</li> <li>2. “start menu”: there are two buttons, “Function” and “Arithmetic”. To confirm the selection, the user will press “BTNC”, and subsequently the users are brought to the respective menus. (image 2)</li> <li>3. “function menu”: there are two buttons, “Table” or “Integration”. (image 3)</li> <li>4. Throughout the menus, the user can toggle SW[15] to switch between “Holding” and “Go-Back” mode. The <b>Seven Segment Display</b> shows outputs depending on the current menu and SW[15].</li> <li>5. “Go-Back” not on in “main menu”: the letters “HOLD” flashes on and off.</li> <li>6. “Go-Back” on in “main menu”: a back-arrow flashes on and off. (image 4)</li> <li>7. “Arithmetic”, “Function”, “Graph” and “Integration”: the letters “ARTH”, “FUNC”, “PLOT” and “INTG” are shown respectively. (image 5)</li> <li>8. For “Arithmetic”, “Function”, “Graph”, “Integration”, if SW[15] is on, it will flash between the back-arrow and the corresponding letters. (image 4)</li> </ol>	 <p><u>Image 1: SYStor menu</u></p>  <p><u>Image 2: Start menu</u></p>  <p><u>Image 3: Function menu</u></p>  <p><u>Image 4: Back arrow and hold</u></p>  <p><u>Image 5: seven segment display showing the mode</u></p>
Student B: Andy Cheng "Fixed-Point Arithmetic, Polynomial Computations + Table, User I/O"	<ol style="list-style-type: none"> <li>1. <b>Fixed-Point Arithmetic Implementation:</b> <ol style="list-style-type: none"> <li>a. Developed the complete fixed-point arithmetic functionality, encompassing both the <b>backend</b> calculation engine and the <b>frontend</b> user interface.</li> <li>b. Implemented the arithmetic_backend module for efficient <b>Q16.16 fixed-point arithmetic operations</b>.</li> <li>c. Created frontend modules for <b>keypad rendering, input/result display, cursor control, and state transition management</b> (image 1)</li> <li>d. Users can navigate the rendered keypads with the on-board buttons (<b>BTNL</b>, <b>BTND</b>, etc) to input numbers, select operations, and view the computed result. (image 2)</li> </ol> </li> <li>2. <b>Polynomial Table Implementation:</b> <ol style="list-style-type: none"> <li>a. Developed the complete polynomial table functionality, integrating <b>backend</b> calculations with a user-friendly <b>frontend</b> interface.</li> </ol> </li> </ol>	 <p><u>Image 1: Numeric and operand keypads</u></p>  <p><u>Image 2: Rendered result/input</u></p>

	<ul style="list-style-type: none"> <li>b. Implemented a dedicated polynomial computation backend module to <b>sequentially calculate y-values</b> using <b>fixed-point operations</b>.</li> <li>c. Implemented <b>table rendering</b> to display <b>calculated x/y value pairs</b>.(image 3)</li> <li>d. Developed <b>user input handling</b> for entering the initial 'x' value via keypad and buttons.</li> <li>e. Enabled <b>table navigation</b> allowing users to scroll through different 'x' value ranges using button inputs.</li> <li>f. Users can input the starting 'x' value or scroll through the table using the on-board buttons (<b>BTNL</b>, <b>BTND</b>, etc).</li> </ul> <p>3. <b>Sequential Fixed-Point to String Conversion:</b></p> <ul style="list-style-type: none"> <li>a. Implemented the <code>fp_to_string_sequential</code> module using a finite state machine (FSM) approach, <b>transforming fixed-point numbers into human-readable strings</b> efficiently.</li> </ul> <p>4. <b>Unified Input Builder for BCD to Fixed-Point Conversion:</b></p> <ul style="list-style-type: none"> <li>a. Developed the <code>unified_input_bcd_to_fp_builder</code> module to <b>convert user-entered BCD values</b> from the keypad into <b>fixed-point numbers, handling negative inputs and backspace</b> for robust input across different modes.</li> </ul>	 <p><u>Image 3: Rendered polynomial table of values</u></p>
<b>Student C:</b> <b>Daniel Kwan</b> "Numerical Integration, OLED Inversion"	<ul style="list-style-type: none"> <li>1. Developed the complete <b>definite integral calculation</b> functionality with <b>decimal and signed support</b>.</li> <li>2. Implemented the <code>integral_computation.v</code> backend module to sequentially calculate the definite integral of the polynomial between <b>user-defined bounds</b>. This uses <b>fixed-point arithmetic</b> based on the fundamental theorem of calculus.</li> <li>3. Integrated backend computation with frontend user input and display management. (image 1)</li> <li>4. Structured the flow so users first enter the lower bound, press "=" to confirm, then enter the upper bound (positive or negative, 0–9, with floating point), and press "=" again to start the computation.</li> <li>5. Implemented frontend modules for:           <ul style="list-style-type: none"> <li>a. User input handling allows users to enter the <b>lower and upper integration bounds</b> via the keypad and buttons. (image 2)</li> <li>b. Result display: presenting the final calculated integral value on the other OLED screen after computation is complete (up to 8 characters is shown, including decimal point) (image 3)</li> </ul> </li> <li>6. Enabled <b>coordination with the graph display</b> module by outputting the user-defined bounds for <b>visualizing the integrated area</b>.</li> <li>7. <b>OLED Visual Inversion:</b> Created a module to rotate the left-side OLED display due to reversed pin alignment on the Basys3 board. (image 5)</li> </ul>	 <p><u>Image 1: Numeric keypad</u></p>  <p><u>Image 2: Input integral bounds</u></p>  <p><u>Image 3: Shaded region with integral result</u></p>  <p><u>Image 5: Before and after rotating left OLED</u></p>
<b>Student D:</b> <b>Ho Wei Hao</b> "Graph Display, Sprite / Character Display"	<ul style="list-style-type: none"> <li>1. The graph module supports plotting of polynomial functions up till the 3rd order. Functions can be plotted even if the coefficients are <b>decimals</b>.</li> <li>2. When <b>SW[14]</b> is turned on, <b>BTNL</b> and <b>BTNR</b> zooms in/out of graph <b>along the x-axis</b> respectively, while <b>BTNU</b> and <b>BTND</b> zooms in/out <b>along the y-axis</b>. There are a total of 4 "zoom-in" states (x1, x2, x4, x8). (image 2)</li> <li>3. Users can pan up to <b>200 times in each direction</b> using <b>BTNU</b>, <b>BTND</b>, <b>BTNL</b> and <b>BTNR</b> to view the full graph. The graph can be panned in the zoomed state. (image 3)</li> <li>4. <b>BTNC</b> is pressed to toggle between zoom and pan operations.</li> <li>5. Turn off <b>SW[14]</b> to resume menu operations.</li> <li>6. When integral is calculated in the subsequent function, the graph will be displayed again, this time displaying the <b>shaded area</b> within the integral bounds chosen by the user. (image 4)</li> <li>7. Implemented sprite and string renderer for <b>standardized character drawing</b> (alphabets, numbers and symbols) across modules.</li> </ul>	 <p><u>Image 1: Plotted cubic function</u></p>  <p><u>Image 2: Zoomed into x-axis independent of y-axis</u></p>  <p><u>Image 3: Panning while zoomed</u></p>

## References

## 1. Character Sprite Library

<https://chatgpt.com/>

The idea of using ROM to standardise character drawing (words, numbers and operations) came from ChatGPT. ChatGPT supplied the foundational code to build the sprite library, and the module to render each library. String renderer was also supplied on top of the sprite render to draw entire words and larger numbers at once.

Both renderers were optimised to reduce hardware resource usage. The supplied code was built upon to provide flexibility of colours, display location, and characters.

Saved memory full ◉  
I am thinking of making a sprite library in verilog, to render 8 by 12 pixels characters like 0 to 9, +, -, \*, and all the capital alphabets. Can you help me generate the foundational code?

That sounds like a fun and useful project, especially if you're working with an FPGA for display rendering or embedded graphics. Here's how we can start:

## Plan

You'll need to:

1. Define a font ROM: Store each character as an 8x12 bitmap.
2. Map character codes to the correct bitmap.
3. Expose an interface to read a particular row of a character bitmap, given a character and a row index (0 to 11).

---

## Character ROM Module (Foundational Code)

This Verilog module:

- Takes an ASCII character and a row index (0-11).
- Outputs an 8-bit row of the corresponding character's bitmap.

### 1. Character ROM Template

```
verilog
module char_rom (
    input [7:0] char_code, // ASCII code of character (e.g. 'A' = 65)
    input [11:0] row_index, // Row Index (0 to 11)
    output reg [7:0] row_data // 8-bit row data (each bit = pixel on/off)
);
begin
    always @(*)
        begin
            case (char_code)
                '0': row_data = $00000000;
                '1': row_data = $00000000;
                '2': row_data = $00000000;
                '3': row_data = $00000000;
                '4': row_data = $00000000;
                '5': row_data = $00000000;
                '6': row_data = $00000000;
                '7': row_data = $00000000;
                '8': row_data = $00000000;
                '9': row_data = $00000000;
                'A': row_data = $00000000;
                'B': row_data = $00000000;
                'C': row_data = $00000000;
                'D': row_data = $00000000;
                'E': row_data = $00000000;
                'F': row_data = $00000000;
                'G': row_data = $00000000;
                'H': row_data = $00000000;
                'I': row_data = $00000000;
                'J': row_data = $00000000;
                'K': row_data = $00000000;
                'L': row_data = $00000000;
                'M': row_data = $00000000;
                'N': row_data = $00000000;
                'O': row_data = $00000000;
                'P': row_data = $00000000;
                'Q': row_data = $00000000;
                'R': row_data = $00000000;
                'S': row_data = $00000000;
                'T': row_data = $00000000;
                'U': row_data = $00000000;
                'V': row_data = $00000000;
                'W': row_data = $00000000;
                'X': row_data = $00000000;
                'Y': row_data = $00000000;
                'Z': row_data = $00000000;
                default: row_data = $00000000;
            endcase
            // Add more characters like '0'-'9' for 'A', etc.
            default: row_data = $00000000;
        end
    end
endmodule
```

This Verilog module:

- Takes an ASCII character and a row index (0-11).
- Outputs an 8-bit row of the corresponding character's bitmap.

### 1. Character ROM Template

### 2. How to Add More Characters

Repeat the `case (row_index)` block for each character you want to support:

- 'A' (65), 'B' (66), ..., 'Z' (90)
- '0'-'9' (48-57)
- '.,~,.,~,' (43, 45, 42, 47)

### 3. Tips

- Design your bitmaps on a grid grid. You can use a tool like a Bx12 bitmap editor online or even draw on graph paper and translate rows into binary.
- You can also store this in a .txt file for easier editing and load it with `readmemh`.

Would you like me to help generate the full ROM contents for all characters in a script or generate a few more examples to get started quickly?

I need a module to render each character

Perfect — you're moving into the rendering module, which uses the `char_rom` to display a character on screen. This module will determine when to turn on each pixel, based on:

- the current screen pixel (`x, y`)
- the position where the character is supposed to be drawn (`char_x, char_y`)
- the `char_code` (i.e. ASCII code like 65 for 'A')

Let me walk you through a foundational Verilog character renderer module.

## char\_renderer.v — Single Character R ↴ Jerer

This module:

### 1. Character ROM Module (Foundational Code)

This Verilog module:

- Takes an ASCII character and a row index (0-11).
- Outputs an 8-bit row of the corresponding character's bitmap.

### 2. How to Add More Characters

Repeat the `case (row_index)` block for each character you want to support:

- 'A' (65), 'B' (66), ..., 'Z' (90)
- '0'-'9' (48-57)
- '.,~,.,~,' (43, 45, 42, 47)

### 3. Tips

- Design your bitmaps on a grid grid. You can use a tool like a Bx12 bitmap editor online or even draw on graph paper and translate rows into binary.
- You can also store this in a .txt file for easier editing and load it with `readmemh`.

Would you like me to help generate the full ROM contents for all characters in a script or generate a few more examples to get started quickly?

I need a module to render each character

Perfect — you're moving into the rendering module, which uses the `char_rom` to display a character on screen. This module will determine when to turn on each pixel, based on:

- the current screen pixel location (`x, y`)
- the position where the character is supposed to be drawn (`char_x, char_y`)
- the `char_code` (i.e. ASCII code like 65 for 'A')

Let me walk you through a foundational Verilog character renderer module.

## char\_renderer.v — Single Character R ↴ Jerer

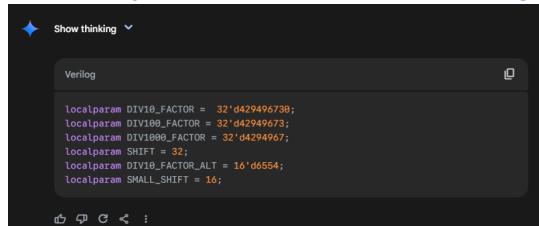
This module:

```

module string_renderer #(
    parameter STR_LBN = 16
)
(
    input [8:0] str_u,
    input [7:0] str_v[STR_LBN:0], // ASCII string
    output reg pixel_x[15:0]
);
parameter CHW_WDTH = 8;
parameter CHW_WDTM = 16;
wire [10:0] row_index;
wire [7:0] row_data;
wire [10:0] col_index;
wire [7:0] current_char;
integer i;
reg found;
assign row_index = screen_y - str_v[0];
char_row_FontRow (
    .char_code(current_char),
    .row_index(row_index),
    .row_data(row_data)
);
always @(*) begin
    pixel_x = 0;
    found = 0;
    for (i = 0; i < CHW_WDTH; i = i + 1) begin
        if (row_index >= 0 & row_index < CHW_WDTH) begin
            screen_x := str_v[i] + 1 * CHW_WDTM;
            screen_x < str_v[i] + (i + 1) * CHW_WDTM;
            screen_x > str_v[i] + CHW_WDTM;
            if (screen_x >= str_v[i] + CHW_WDTM) begin
                current_char = str[1];
                col_index = screen_x - (str_v[i] + i * CHW_WDTM);
                pixel_x = row_data[7 - col_index];
                found = 1;
            end
        end
    end
end
end

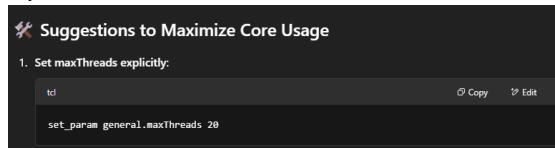
```

2. Inspiration to Perform Fixed-Point Arithmetic in Q16.16  
<https://projectcf.io/posts/fixed-point-numbers-in-verilog/>
3. Using a Multiplication and Shift Based Approach for Division  
<https://projectcf.io/posts/division-in-verilog/>



Magic numbers (corresponding to approximately  $2^{32}/10$  and so on) from Google Gemini. These values were crucial in our optimized division approach which led to resource savings necessary for our program to compile.

4. Optimization Advice



[https://docs.amd.com/r/en-US/ug901-vivado-synthesis/USE\\_DSP](https://docs.amd.com/r/en-US/ug901-vivado-synthesis/USE_DSP)

Sources explored to minimise resource usage (LUT) for our program to compile. The fix provided in AMD's documentation allowed for us to better balance between DSP and LUT consumption.

## Feedback

1. The labs were planned well and the difficulty curve increased smoothly.