Image Calculator

CS152B Digital Design Project Laboratory

TA: Babak Moatamed
Students: Jonathan Hurwitz, Alex Peters, Vilius Vysniauskas

I. Introduction

Our aim was to create a calculator in MicroBlaze that can ”see” inputs and evaluate an expression for the user. By using image processing in order to detect numbers and operators (+, -, /, *), our calculator can evaluate basic expressions and then output the result to the user by writing to the console over a serial connection. The use case is as follows:

1) The user turns on the calculator and waits for the camera to initialize. The initialization sequence is complete when the red box appears on the screen.
2) To capture a number or operator, place it within the view of the red box. Flip the first switch to capture the number. Repeat this step until your entire expression has been captured.
3) To clear an expression, flip the third switch. The result can be displayed by flipping the second switch.

The calculator currently only supports four digit operands and a single operator per expression, but this can be changed by adjusting the size of our character buffer. Erroneous corner cases such as back to back operators, starting with an operator, and divide by zero were handled. Relevant error output messages are displayed to the user when one of these cases is encountered.

II. Project Background

Our final project began at a much different starting point than where it is now. Initially, we intended to combine multiple hardware modules to create a semi-autonomous, color-retrieving robot. To accomplish this we wanted to integrate both the provided iRobot and VMODCam modules in order to build a iRobot that can see its surroundings, detect one of several colors, move towards the color, and come back to its starting point (in essence retrieving the color).

We set out by creating a new EDK project based purely on the VMODCam module. After importing the necessary files, updating the necessary IP cores (an automatic process done by the Xilinx Platform Studio), and waiting for the long export, we had a working hardware design with all the correct peripherals (mainly HDMI, DVMA, two IICs, and camera controllers). Following the export to the SDK, we continued with the tutorial by assembling a project structure and inserting the tutorial code.

Programming the FPGA and running the project for the first time left us very confused. The screen, connected with HDMI, outputted two adjacent green gradients and nothing else. This was perplexing but we soon realized that the cameras needed a long period of time (roughly 5 to 7 minutes) to initialize and begin a real-time feed to the monitor. Furthermore, we also discovered that the tutorial code for the camera initialization contained useful progress and debugging statements. To see these, we connected the FPGA to a serial port and listened to the appropriate COM channel.

With this quick success, we decided to integrate the iRobot hardware into the existing project so that we would not have to worry about breaking anything later on (this was naive thinking in hindsight). The iRobot tutorial required a different UART peripheral (a xps_uart16650 as opposed to a xps_uartlite), along with two GPIO peripherals, for LED and push button support, and an interrupt controller. After re-mapping addresses and successfully exporting the design, we ran into our first major hurdle - the SDK was simply unable to link the included libraries in our C code. A week and a half later, we noticed that our interrupt controller did not have the correct output pin configured, and even though the hardware design did not fail to export, this was the cause of the linker failure in the SDK.

Another obstacle tackled, we regained motivation and moved forward, only to quickly crash into another Xilinx barrier. Although the code was now linking and compiling, the two different code examples interfered with each other. For example, if we ran the camera code first, the camera would work but the iRobot code would not execute with the program seemingly hanging forever (debugging was not helpful - the program executed random jumps to weird memory locations).

If we put the iRobot code first, the iRobot code would successfully execute the provided basic commands but the camera initialization steps would hang on register write instructions. After additional inspection, we came to the conclusion that the two different UART peripherals were the cause (since the camera tutorial specified two of the same type whereas the iRobot tutorial only needed one).

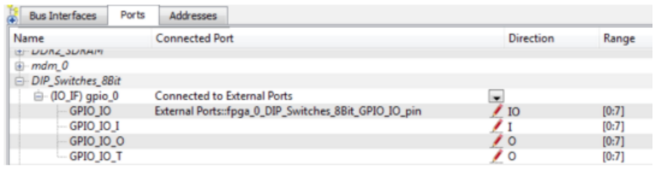
At this point in time, 2 and a half weeks after starting, we could not spend more time debugging our hardware design while still expecting to finish. Because of this time pressure, we decided that we would continue only working with the VMODCam module in order to avoid the debugging nightmare of attempting to fuse two different peripheral configurations. As we brainstormed new ideas to implement with a sole camera module, we agreed upon a functional and useful image processing project.

The image processing calculator uses the Virtex-5 FPGA with Digilent Genesys development board. The VMOD camera module was used in order to capture images. Data was carried over the VHDCI cable and images were displayed on the monitor via an HDMI to DVI cable. Serial data was read from the FPGA via a serial to USB type A converter cable.

III. Tutorial

Our final project is based on the provided VMODCam hardware design configuration and sample code (on CCLE). We used a Xilinx Virtex 5 FPGA board. Here are the steps we took:

1) Download and unzip the VMOD Camera Tutorial.zip archive from CCLE
    - This location should contain four folders: binaries, doc, IPRepository, and proj.
    - The binaries folder has a bit file, but this tutorial will generate its own
    - The IPRepository folder holds the peripherals you need to add to the XPS project to connect the VmodCam to the FPGA board
    - The proj folder holds several test projects, as well as system.xmp (which will be our starting point)
2) Open proj/system.xmp in Xilinx XPS
    - Go through the IP update process if prompted, updating all IP cores
3) Add a GPIO module, for DIP switches
    - Connect it to external ports - assigned to GPIO_IO (as shown below)



Name	Connected Port	Direction	Range
DDR2_SDRAM			
mdm_0			
DIP_Switches_8Bit			
(IO_IF) gpio_0	Connected to External Ports		
GPIO_IO	External Ports::fpga_0_DIP_Switches_8Bit_GPIO_IO_pin	IO	[0:7]
GPIO_IO_I		I	[0:7]
GPIO_IO_O		O	[0:7]
GPIO_IO_T		O	[0:7]

Figure 1: An example of the ports configuration.

    - Set the UCF file correctly, matching up the named peripheral pins with the labelled physical pins on the FPGA board

4) Export the design to SDK
5) In the SDK, create a new workspace and an empty Application Project inside it
6) Import (as a filesystem) the tutorial code from the following directory: proj/TestApp_VmodCam/src
    - You should now have five files under your projects source (src) folder: cam_ctrl_header.h , cam_ctrl.c , main.c , vmodcam_cfg.h , vmodcam_header.h
7) Change memory region mapping
    - Go to the project lscript.ld file
    - Change all from ddr2_sdram_MPMC_BASEADDR to ilmb_cntlr_dlmb_cntlr
8) Make sure to setup FPGA board jumpers correctly
    - The VSWT x VUEWP (JP1) jumpers on the right hand side in between the two VHDCI ports on the board should be bridged
    - The M2 voltage mode jumpers on the upper right of the board should be bridged
    - No other mode (M1 or M0) should be connected
    - Other jumpers on the board should be ok at their default states
9) Connect and configure necessary cables. The figure below shows (from left to right) the VHDCI, HDMI to DVI, power, USB, and serial cable connected to the Genesys breakout board:

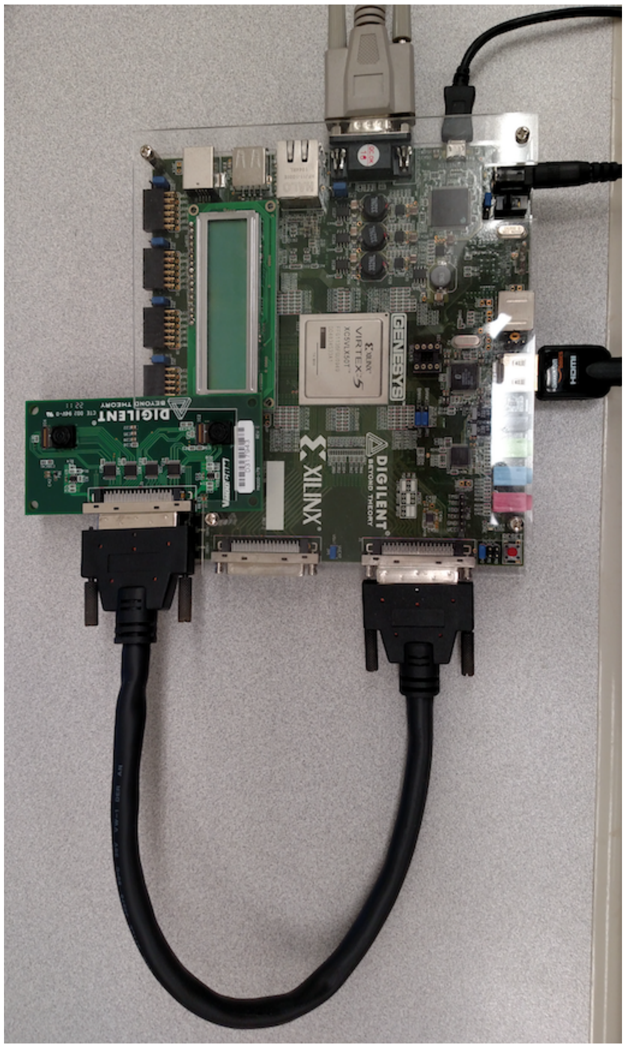
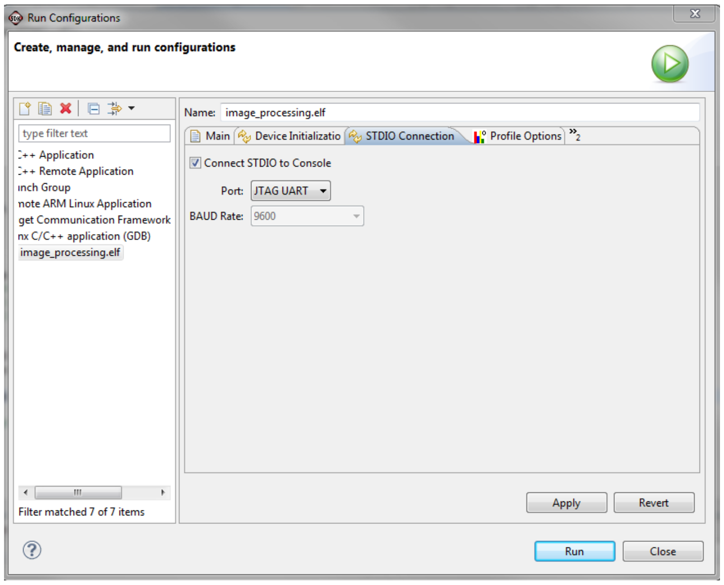


Figure 2: The fully connected system with VMOD camera, HDMI to DVI cable, DC-in, USB, and serial.

- Connect serial cable (to read STDOUT debug statements)
  - You may need to obtain an adapter from serial to USB in order to connect it to the computer which is running the SDK
  - Enable the SDK terminal to output the serial port, as shown below:



Name: image_processing.elf

☑ Connect STDIO to Console

Port: JTAG UART

BAUD Rate: 9600

Figure 3: "Run configurations" window allowing user to connect STDIO to the console. Options to configure port and Baud rate as well.

  - Make sure to select the correct baud rate (115200 if VMODCam module) and device (for us it was COM3)
  - It is also possible to do this with the PUTTY program, but make sure that only one is connected to the COM port at a time
- Connect all other cables
  - HDMI to desired monitor for camera output
  - USB from computer which contains the project code (to upload to FPGA)
  - DC power cable to power brick

10) Program the FPGA (be extra careful to select the correct .bit and .bmm files)
11) Run the project on hardware (from the menu bar)
    - This should be enough to get basic camera output working
    - Two green and adjacent gradients should appear as debug statements begin to show up on the terminal (either in SDK or in Putty depending on previous steps)
    - The two cameras take about 5 minutes to initialize, so be patient
    - The last debug output: Setting up Camera Control Block successfully!
12) Tips & Comments
    - To move the second camera output off the screen, change (in main.c): CamCtrlInit(XPAR_CAM_CTRL_1_BASEADDR, MODE, 0) to

CamCtrlInit(XPAR_CAM_CTRL_1_BASEADDR, MODE, 2560)

- The cameras support three different modes: 640x480p, 800x600p, 1280x720p
  - There are three defines associated with them:
    #define CAM_CFG_640x480P 0x0001
    #define CAM_CFG_800x600 0x0002
    #define CAM_CFG_1280x720P 0x0004
  - To change modes, modify the following lines (in main.c):
    CamIicCfg(XPAR_CAM_IIC_0_BASEADDR, MODE_DEFINE);
    CamIicCfg(XPAR_CAM_IIC_1_BASEADDR, MODE_DEFINE);
    CamCtrlInit(XPAR_CAM_CTRL_0_BASEADDR, MODE_DEFINE, 1280);
    CamCtrlInit(XPAR_CAM_CTRL_1_BASEADDR, MODE_DEFINE, 0);
    Note: Put the desired mode in the MODE_DEFINE parameter
- The monitor output will be vertically stretched compared to the captured image
  - For example, if we are capturing in the 1280x720p mode, the image will be displayed in stretched manner on a 1280x1024 monitor (in our case)

IV. Design

A. main function

The core logic is implemented in "main.c", and consists of a main function as well as a helper function "printAnswer". The code for the main function is shown below, broken into several blocks for ease of explanation:

```
int main() {
        u32 lDvmaBaseAddress = XPAR_DVMA_0_BASEADDR;
        int posX, posY;

        // Initialize switches
        XGpio_Initialize(&switches, XPAR_DIP_SWITCHES_8BIT_DEVICE_ID);
        XGpio_SetDataDirection(&switches, 1, 7);

        for(posX = 0; posX<2560; posX++)
                for(posY = 0; posY<720; posY++) // 720
                        XIo_Out16(XPAR_DDR2_SDRAM_MPMC_BASEADDR +
                        2*(posY*2560+posX), 0);

        for(posX = 0; posX<2560; posX++)
                for(posY = 0; posY<720; posY++) // 720
                        XIo_Out16(XPAR_DDR2_SDRAM_MPMC_BASEADDR +
                        2*(posY*2560+posX), (posX/40)<<4);

        XIo_Out32(lDvmaBaseAddress + blDvmaHSR, 40);     // hsync
        XIo_Out32(lDvmaBaseAddress + blDvmaHBPR, 260);   // hbpr
        XIo_Out32(lDvmaBaseAddress + blDvmaHFPR, 1540);  // hfpr
        XIo_Out32(lDvmaBaseAddress + blDvmaHTR, 1650);   // htr
        XIo_Out32(lDvmaBaseAddress + blDvmaVSR, 5);      // vsync
        XIo_Out32(lDvmaBaseAddress + blDvmaVBPR, 25);    // vbpr
        XIo_Out32(lDvmaBaseAddress + blDvmaVFPR, 745);   // vfpr
        XIo_Out32(lDvmaBaseAddress + blDvmaVTR, 750);    // vtr

        XIo_Out32(lDvmaBaseAddress + blDvmaFWR, 0x00000500); // frame width 0x00000500
        XIo_Out32(lDvmaBaseAddress + blDvmaFHR, 0x000002D0); // frame height 0x000002D0
        XIo_Out32(lDvmaBaseAddress + blDvmaFBAR, XPAR_DDR2_SDRAM_MPMC_BASEADDR); // frame base ad
        XIo_Out32(lDvmaBaseAddress + blDvmaFLSR, 0x00000A00); // frame line stride 0x00000A00
        XIo_Out32(lDvmaBaseAddress + blDvmaCR, 0x00000003); // dvma enable, dfl enable

        CamIicCfg(XPAR_CAM_IIC_0_BASEADDR, 0);
        CamIicCfg(XPAR_CAM_IIC_1_BASEADDR, 0);
        CamCtrlInit(XPAR_CAM_CTRL_0_BASEADDR, 0, 2560);
        CamCtrlInit(XPAR_CAM_CTRL_1_BASEADDR, 0, 0);
```

The code block above is responsible for camera configuration and initialization. This code was provided in the VMOD camera tutorial files. We were able to obtain a full size image on the screen by shifting the second camera off by 2560 pixels, shown in the first "CamCtrlInit". Also, we learned that by enabling both cameras, the exposure levels were much better than by enabling just one. This led us to believe that there is some sort of mixing going on in the data stream rather than discrete channels for each camera.

```

        u8 switch_val = 0;
        u8 capture_data = 0;
        u8 old_capture_data = 0;
        u8 count = 0;

        u8 num_vals = 0;
        u8 get_answer = 0;
        u8 clear = 0;
        u8 has_been_cleared = 1;

        short captured_chars[5];
        char expression[EXPRESSION_SIZE]= {0};
        char op = '$';

        while(1) {
                for(posX = 530; posX<751; posX++){
                        for(posY = 250; posY<261; posY++){
                                XIo_Out16(XPAR_DDR2_SDRAM_MPMC_BASEADDR +
                                 2*(posY*2560+posX), 0xf00);
                        }
                }
                for(posX = 530; posX<751; posX++){
                        for(posY = 460; posY<471; posY++){
                                XIo_Out16(XPAR_DDR2_SDRAM_MPMC_BASEADDR +
                                 2*(posY*2560+posX), 0xf00);
                        }
                }

                for(posX = 530; posX<541; posX++){
                        for(posY = 260; posY<461; posY++){
                                XIo_Out16(XPAR_DDR2_SDRAM_MPMC_BASEADDR +
                                 2*(posY*2560+posX), 0xf00);
                        }
                }

                for(posX = 740; posX<751; posX++){
                        for(posY = 260; posY<461; posY++){
                                XIo_Out16(XPAR_DDR2_SDRAM_MPMC_BASEADDR +
                                 2*(posY*2560+posX), 0xf00);
                        }
                }
        }
```

The while loop above continuously redraws the four rectangles that make up our red box in the center of the screen.

```

        old_capture_data = capture_data;

        switch_val = XGpio_DiscreteRead(&switches, 1);
        capture_data = switch_val & 0x1;
        get_answer = switch_val & 0x2;
        clear = switch_val & 0x4;

        if (clear && !has_been_cleared) {
                op = '$';
                num_vals = 0;
                capture_data = 0;
                old_capture_data = 0;
```

```
14          int i = 0;
15          for (; i < EXPRESSION_SIZE; i++)
16              expression[i] = 0;
17          xil_printf(”Your expression has been cleared.\n\r”);
18          has_been_cleared = 1;
19      }
20      // Switch OFF −> ON
21      else if(!old_capture_data && capture_data){
22          // Start processing the grid blocks
23          count = 0;
24          captured_chars[0] = processImage(540, 260);
25          count++;
26      }
27      // Switch ON −> ON
28      else if (old_capture_data && capture_data) {
29          if (count < 5) {
30              captured_chars[count] = processImage(540, 260);
31              count++;
32          }
33      }
34      // Switch ON −> OFF
35      else if (old_capture_data && !capture_data) {
36          char result = determineChar(captured_chars, count);
37          if (result == ’?’) {
38              xil_printf(”Please re−enter number or operator, unable to
39              scan correctly.\n\r”);
40              continue;
41          }
42          if (result == ’+’ || result == ’−’
43           || result == ’*’ || result == ’/’) {
44              if (op == ’$’) {
45                  if (num_vals == 0) {
46                      xil_printf(”First input cannot
47                      be an operator.\n\r”);
48                      continue;
49                  }
50                  op = result;
51              }
52              else {
53                  xil_printf(”Cannot have multiple
54                   operators in an expression.\n\r”);
55                  continue;
56              }
57          }
58          expression[num_vals] = result;
59          num_vals++;
60
61          xil_printf(”%s\n\r”, expression);
62          has_been_cleared = 0;
63      }
64      // Compute answer
65      else if (get_answer && (num_vals > 2)) {
66          expression[num_vals] = ’\0’;
67          printAnswer(expression, num_vals, op);
68
69          capture_data = 0;
70          old_capture_data = 0;
71          op = ’$’;
72          num_vals = 0;
73          int i = 0;
74          for (; i < EXPRESSION_SIZE; i++)
75              expression[i] = 0;
76          xil_printf(”Your expression has been cleared.\n\r”);
77          }
78      }
79
80      return 0;
81  }
```

The conditional statements shown above determine actions based on user dip switch input. Line 37 shows a check to make sure that the image processing code was able to determine an actual operand or operator. A ’?’ is returned when the function cannot determine a correct value. The next block checks to make sure that the first input is not an operator. The check for two back to back operators is done in the else fallthrough.

If the corner cases pass, the answer is computed for the user and the expression is automatically cleared so that the user can perform another operation. Flags are reset for the next iteration of the outer loop.

Shown below is the ”printAnswer” function, which takes care of divide-by-zero and unknown operator cases, although the latter should never happen (due to our check in the main function):

```
void printAnswer(const char* expression, u8 num_elem, char operator) {
    int a = 0, b = 1, result = 0;
    char err = 0;

    switch (operator) {
        case ’+’:
            stupidscanf(expression, &a, &b);
            result = a+b;
            break;
        case ’−’:
            stupidscanf(expression, &a, &b);
            result = a−b;
            break;
        case ’*’:
            stupidscanf(expression, &a, &b);
            result = a*b;
            break;
        case ’/’:
            stupidscanf(expression, &a, &b);
            if (b == 0) {
                xil_printf(”Cannot divide by zero.\n\r”);
                result = 0;
                err = 1;
            }
            else
                result = a/b;
            break;
        default:
            xil_printf(”Error: unknown operator.\n\r”);
            err = 1;
    }
    if (!err)
        xil_printf(”Your answer is: %d\n\r”, result);
}
```

*B. void calcThreshold(int xBase, int yBase, short *thresholdRGB)*

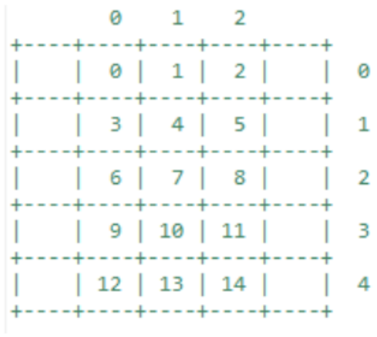
This function calculates the RBG color values of the white background. The sample is taken from a block to the top right of the scanning area. For each pixel in the sample block, we read the RGBx444 value from DDR2. We then use bitmasks to extract the correct values for red, green, and blue and add each to a sum for that color. After all the values have been added, the average color is computed by dividing by the number of pixels in the sample block. Finally, the averages are stored in a short array, threshholdRGB.

```
void calcThreshold(int xBase, int yBase, short *thresholdRGB) {
    int xStart = xBase−60; // the +1 is to avoid the red boxes, not sure if needed
    int yStart = yBase−60; // ^^
    int posX, posY;
    int xEnd = xStart + GRID_WIDTH;
    int yEnd = yStart + GRID_LENGTH;

    // Average the pixel color of the block
    short numPixels = GRID_WIDTH * GRID_LENGTH;
    short avgR = 0;
    short avgG = 0;
    short avgB = 0;
    u16 color = 0;

    for (posY = yStart; posY < yEnd; posY++) {
        for (posX = xStart ; posX < xEnd; posX++) {
            color = XIo_In16(XPAR_DDR2_SDRAM_MPMC_BASEADDR + 2*(posY*2560+posX));
            avgB += (color&0x000f);
            avgG += ((color&0x00f0)>>4);
            avgR += ((color&0x0f00)>>8);
        }
    }

    avgR /= numPixels;
```

```
        avgG /= numPixels;
        avgB /= numPixels;

        thresholdRGB[0] = avgR;
        thresholdRGB[1] = avgG;
        thresholdRGB[2] = avgB;
}
```

*C. int processGridBlock(char blockno, int xBase, int yBase, short *thresholdRGB)*

The grid block structure is shown below:



```
       0    1    2
+----+----+----+----+----+
|    | 0  | 1  | 2  |    |  0
+----+----+----+----+----+
|    | 3  | 4  | 5  |    |  1
+----+----+----+----+----+
|    | 6  | 7  | 8  |    |  2
+----+----+----+----+----+
|    | 9  | 10 | 11 |    |  3
+----+----+----+----+----+
|    | 12 | 13 | 14 |    |  4
+----+----+----+----+----+
```

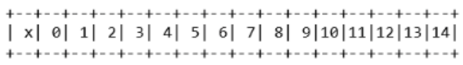
Figure 4: Grid block structure.

This function uses the same process as calcThreshold, but instead of determining the white threshold, it determines the average color value of one of the 15 grid blocks. After the averages are computed, they are compared against the precomputed RGB thresholds. If at least two of the three averages are below their respective threshold, then we determine that there is a line inside that grid block, and return a 1. Otherwise, we determine that block is empty and thus return a 0.

```
int processGridBlock(char blockno, int xBase, int yBase, short *thresholdRGB) {
        int xStart = blockno % 3;
        int yStart = 0;
        if (blockno < 3) yStart = 0;
        else if (blockno < 6) yStart = 1;
        else if (blockno < 9) yStart = 2;
        else if (blockno < 12) yStart = 3;
        else yStart = 4;

        int posY = yBase + yStart*GRID_LENGTH;
        int yEnd = posY + GRID_LENGTH;
        int posX = xBase + GRID_WIDTH + xStart*GRID_WIDTH;
        xStart = posX;
        int xEnd = posX + GRID_WIDTH;


        // Average the pixel color of the block
        short numPixels = GRID_WIDTH * GRID_LENGTH;
        short avgR = 0;
        short avgG = 0;
        short avgB = 0;
        u16 color = 0;

        for ( ; posY < yEnd; posY++) {
                for (posX = xStart ; posX < xEnd; posX++) {
                        color = XIo_In16(XPAR_DDR2_SDRAM_MPMC_BASEADDR
                        + 2*(posY*2560+posX));

                        avgB += (color&0x000f);
                        avgG += ((color&0x00f0)>>4);
                        avgR += ((color&0x0f00)>>8);
                }
        }

        avgR /= numPixels;
        avgG /= numPixels;
        avgB /= numPixels;

        u8 sum = 0;
        if(avgR < thresholdRGB[0]-2){
                sum += 1;
        }
        if(avgG < thresholdRGB[1]-2){
                sum += 1;
        }
        if(avgB < thresholdRGB[2]-2){
                sum += 1;
        }

        if (sum >= 2)
                return 1;

        return 0;
}
```

D. short processImage(int xBase, int yBase)

This function uses the above functions to scan a 200 x 200 pixel area of the image. First, the white threshold is determined by calcThreshold. Then, processGridBlock is called on each of the 15 blocks shown in Fig. . If processGridBlock returns a 1, a 1 is inserted into the corresponding bit location of the 16-bit bitmap (diagrammed in the figure shown below).



```
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| x| 0| 1| 2| 3| 4| 5| 6| 7| 8| 9|10|11|12|13|14|
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

Figure 5: Structure of a bitmap in memory for our implementation.

When all the blocks have been processed, the bitmap is returned. The code is shown below:

```
short processImage(int xBase, int yBase) {
        short num = 0;
        short threshold[3];
        char n;

        calcThreshold(xBase, yBase, threshold);

        int val;
        for (n = 0; n < NUM_BLOCKS; n++) {
                val = processGridBlock(n, xBase, yBase, threshold);
                if (val) {
                        num |= (BASE_MASK >> n);
                }
        }
        return num;
}
```

*E. char determineChar(short *arr, int len)*

This function takes in an array of 2-byte bitmaps, averages them into a single bitmap, and determines what character that bitmap corresponds to. To do this, we first determine the sum of each position in the bitmap by traversing through each bit in the bitmap and if it is a 1, adding 1 to a running sum for that position. This is repeated for all bitmaps in the array. Once the sums are computed, the average of each position is calculated by dividing each sum by the number of elements in the array. If the average for a position is greater than or equal to 0.5, a 1 is put in that position for the final bitmap. Otherwise, a 0 is put in that position. Once this has been done for all 15 positions, the final bitmap can be used to determine the correct character. For all characters except /, the bitmap is switched against bitmaps that we predetermined to match each character. If the final bitmap does not match any of these, a ? is returned. For detecting /, we check if the top row (blocks 0 - 2) and the bottom row (blocks 12 - 14) are empty. This ensures that we dont detect a number as the division operator. We also make check that blocks 3 and 11 are empty, so that we dont falsely identify a * as a / operator. The last check for division is to ensure blocks 5 and 9 are both filled, which ensures + and - do not trigger the division operator.

```c
char determineChar(short *arr, int len) {
        float sum[15] = {0};
        int i, j;

        for (i = 0; i < len; i++) {
                short bitcode = arr[i];
                for (j = 0; j < NUM_BLOCKS; j++) {
                        if (bitcode & (BASE_MASK >> j)) {
                                sum[j]++;
                        }
                }
        }

        float avg;
        short num = 0;
        short printval;
        for (j = 0; j < NUM_BLOCKS; j++) {
                avg = sum[j] / len;
                if (avg >= 0.5) {
                        num |= (BASE_MASK >> j);
                }
        }

        // Division detection
        if ((~num & MASK_DIV1) == MASK_DIV1 && (num & MASK_DIV2)) {
        // checks if blocks 0 - 3 and 11 -14 rows are not filled
        // && if blocks 5 and 9 are filled
                return '/';
        }

        switch (num) {
                case MASK0: return '0';
                case MASK1: return '1';
                case MASK2: return '2';
                case MASK3: return '3';
                case MASK4: return '4';
                case MASK5: return '5';
                case MASK6: return '6';
                case MASK7: return '7';
                case MASK8: return '8';
                case MASK9: return '9';
                case MASK_ADD: return '+';
                case MASK_MULT: return '*';
                case MASK_SUB: return '-';
                default: return '?';
        }
}
```

F. void stupidscanf(const char expression, int *a, int *b)*

Takes in a C string of the form [number1][operator][number2] and stores number1 and number2 in integers a and b. The function goes through the string from left to right, multiplying the result so far by 10 and adding the new digit in. The operator is used to determine when to start calculating int b.

```c
void stupidscanf(const char* expression, int *a, int *b) {
        int ta = 0, tb = 0, i, state = 1;
        unsigned len = strlen(expression);

        for (i = 0; i < len; i++) {
                if (state) {
                        switch (expression[i]) {
                                case '+':
                                case '-':
                                case '*':
                                case '/':
                                        state = 0;
                                        break;
                                default:
                                        ta *= 10;
                                        ta += (expression[i] - '0');
                                        break;
                        }
                }
                else {
                        tb *= 10;
                        tb += (expression[i] - '0');
                }
        }

        *a = ta;
        *b = tb;
}
```

V. Testing

In the EDK, testing was mainly dealt with by avoiding it. After our project pivot and all the tribulations that it entailed, we decided to be very careful about adding new peripherals. Before exporting, we precisely determined the exact modules that we would need. We added a GPIO peripheral to ensure that we could use the switches as a camera press action, an equation solver action, and a reset action. Using experience from previous projects, this was easy enough to do simply by connecting the correct ports and modifying the UCF file.

The main hardware challenge was a faulty camera module. For certain modes of operation, the camera would get stuck in a no-instruction loop, waiting for certain memory addresses to be correctly set. We spent a lot of time attempting to fix this in software, but in the end, we swapped the physical camera module, and everything began to work. We suspect this had to do with certain registers being non-functional or shorted incorrectly. This fix enabled us to capture in the full 1280x720p mode.

Most testing in the SDK was performed using the xil_printf() function to allow us to use debug statements to see intermediate steps of certain algorithms. In general, our approach was to make several large changes in one file at a time (for example, either in main.c or in determineChar.c but not in both) so as to avoid multiple concurrent points of failure, but also so that each run of the project was not for a single inconsequential change (recall that the cameras take 5 minutes to initialize).

The main challenge that arose was debugging how to convert (scan) a string for a pattern of: number-operator-number. We intended to use the sscanf() function in the STL library but quickly ran into memory issues (with the compiler complaining the project could not fit into the allocated .text region). To fix this problem, we wrote our own version of a simple scanf() function to perform a more optimized and specific portion of the desired sscanf().

For the final presentation, we decided to generate digital representations of the numbers in order to increase digit uniformity. Also, by displaying these on one of the monitors, we got a light intensity increase for free (and did not have to use backlighting).

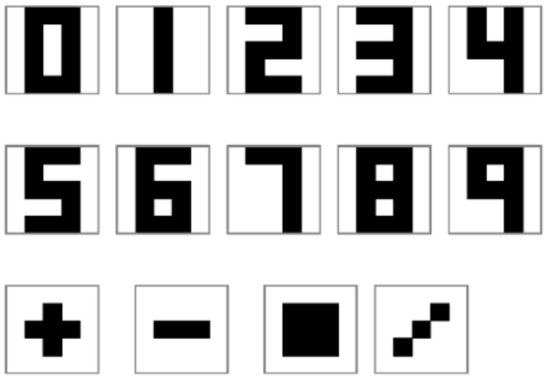


Figure 6: Digital representation of the numbers and operators used for testing and demonstration.

This proved to be very effective for testing and the demo.

VI. Conclusion

Our calculator was able to pick up operators and operands with zero false positives. Edge cases such as divide by zero, leading operators, and back-to-back operators were correctly handled. Many 4-digit number combinations were tested, along with all of the possible operators. The calculator evaluated expressions correctly.

VII. Discussion

Since MicroBlaze is not currently in use or active development by any large organization, the resources are scarce and the support network is weak. It was difficult to find help outside of the classroom for some of our issues. Much of the time we had to use trial-and-error.

We initially planned to draw the letters on paper cards. However, the camera's exposure levels were too low to pick up the whites accurately so we tried different lighting methods. First, we tried to use an x-ray viewer as a backlight, but the 60Hz noise on the florescent bulbs interfered with the image capture. We also tried to build an LED backlight but weren't able to create a good enough diffuser. In true Xilinx/MicroBlaze/Digilent fashion, the camera exposure levels magically fixed themselves so we were able to remove the backlight altogether for testing. We opted to use the extra monitor to display the numbers and operators for the final presentation, since it was easier to maintain digit uniformity.