Edinburgh Napier
University

Programmable Logic Design

ELE10105

PicoBlaze Implementation.

40292302

May 2020

Table of Content:

Introduction

This report describes the implementation of a Reverse Polish Notation (RPN) calculator using the Picoblaze microcontroller core. Picoblaze is a fully embedded 8-bit Reduced Instruction Set Computer (RISC) microcontroller designed to be implemented on Xilinx FPGA devices (Xilinx and Inc., 2011). This allows the controller to be fully customisable and adaptable to future technologies. The application presented in this report is programmed using the pBlazeIDE using a derivative of the assembly language. The report will first introduce the principle of the RPN calculator accompanied by the implementation in pBlaze with annotated code. Then, a simulation report will be given for each of the operations and routines available in the programme. Finally, a short discussion on picoblaze and the work undertaken will be given.

I. Description of Operational Code

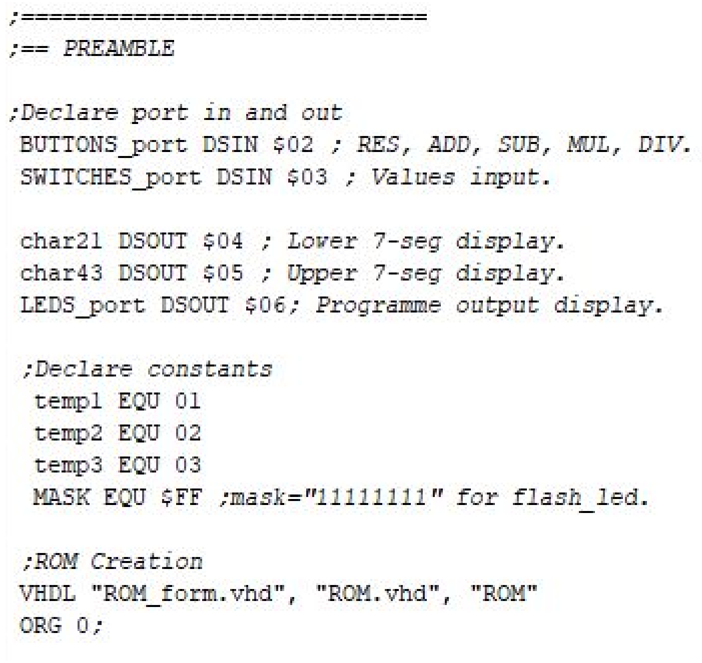
RPN Calculation Operation Description:

Reverse Polish Notation is aimed at avoiding the use of brackets to define the priority of operations in a calculation. The numbers are stacked followed by the operator thus allowing a faster processing of the operations. (Stone, 2018)

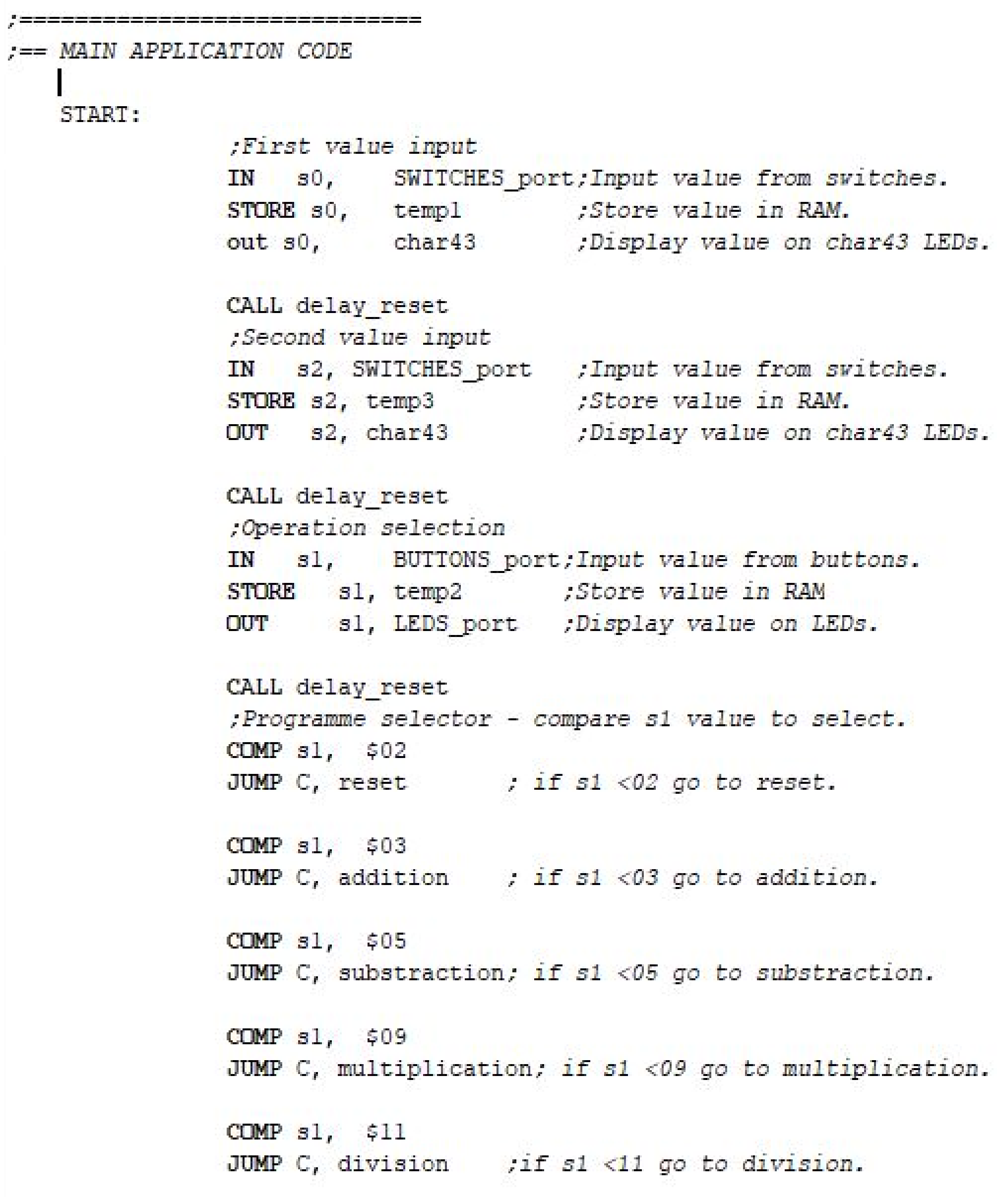
Example: “2 + 3” using RPN becomes “2 3 +”.

Assembly pBlaze - RPN Calculator Programme:

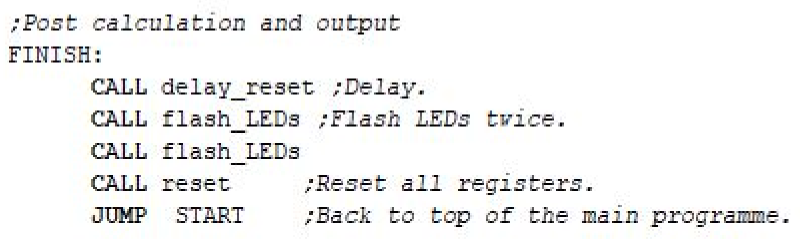
This part presents the various sections in the programme: Preamble, Main Application code and subroutines. The code allows entering the numerical values followed by an operator. The programme will then compute, return the values on a display and reset. LEDs will flash whilst computing and flash 4 times when the final result is displayed.



```
;==============================
;== PREAMBLE

;Declare port in and out
 BUTTONS_port DSIN $02 ; RES, ADD, SUB, MUL, DIV.
 SWITCHES_port DSIN $03 ; Values input.

 char21 DSOUT $04 ; Lower 7-seg display.
 char43 DSOUT $05 ; Upper 7-seg display.
 LEDS_port DSOUT $06; Programme output display.

 ;Declare constants
  temp1 EQU 01
  temp2 EQU 02
  temp3 EQU 03
  MASK EQU $FF ;mask="11111111" for flash_led.

 ;ROM Creation
 VHDL "ROM_form.vhd", "ROM.vhd", "ROM"
 ORG 0;
```



```
;============================
;== MAIN APPLICATION CODE

START:
            ;First value input
            IN   s0,    SWITCHES_port;Input value from switches.
            STORE s0,   temp1        ;Store value in RAM.
            out s0,     char43       ;Display value on char43 LEDs.

            CALL delay_reset
            ;Second value input
            IN   s2, SWITCHES_port   ;Input value from switches.
            STORE s2, temp3          ;Store value in RAM.
            OUT   s2, char43         ;Display value on char43 LEDs.

            CALL delay_reset
            ;Operation selection
            IN   s1,    BUTTONS_port;Input value from buttons.
            STORE    s1, temp2      ;Store value in RAM
            OUT      s1, LEDS_port  ;Display value on LEDs.

            CALL delay_reset
            ;Programme selector - compare s1 value to select.
            COMP s1,  $02
            JUMP C, reset        ; if s1 <02 go to reset.

            COMP s1,  $03
            JUMP C, addition     ; if s1 <03 go to addition.

            COMP s1,  $05
            JUMP C, substraction; if s1 <05 go to substraction.

            COMP s1,  $09
            JUMP C, multiplication; if s1 <09 go to multiplication.

            COMP s1,  $11
            JUMP C, division     ;if s1 <11 go to division.
```



```
;Post calculation and output
FINISH:
      CALL delay_reset ;Delay.
      CALL flash_LEDs ;Flash LEDs twice.
      CALL flash_LEDs
      CALL reset     ;Reset all registers.
      JUMP  START    ;Back to top of the main programme.
```

```
;==============================
;== SUB ROUTINES

delay_reset:
    LOAD  s4, s4               ;Load register with itself to create delay.
    RET

flash_LEDs:
         LOAD s4, $AA          ;Load "10101010" to s4.
         OUT s4, LEDS_port     ;Display s4 on LEDs.
         CALL delay_reset      ;Delay.
         XOR s4, MASK          ;s4 XOR MASK ("11111111").
         OUT s4, LEDS_port     ;Display new s4 on LEDs.
         RET                   ;Back to main programme.
reset:
        ;Reset all registers, displays and RAM:
        LOAD    s0, $00        ;LOAD "00000000" to s0 then load s0 to the rest.
        LOAD    s1, s0
        LOAD    s2, s0
        LOAD    s3, s0
        LOAD    s4, s0
        LOAD    s5, s0
        LOAD    s6, s0
        LOAD    s7, s0
        LOAD    s8, s0
        LOAD    s9, s0
        OUT s0, char21
        OUT s0, char43
        OUT s0, LEDS_port
        JUMP    START          ; reset performed back to programme start.
```

```
substraction:
        SUB s0, s2              ;Substraction calculation.
        CALL flash_LEDs         ;LEDs flashing during calculation.
        OUT s0, char21          ;Output result
        JUMP    FINISH          ;Go to sequence finish.

addition:
        ADD s0, s2              ;Addition calculation.
        CALL flash_LEDs         ;LEDs flashing during calculation.
        OUT s0, char21          ;Output result.
        JUMP    FINISH          ;Go to sequence finish.

division:
        ;Register Declaration:
        ;dividend - s0
        ;divisor - s2
        quotient     EQU s8
        remainder    EQU s9
        bit_mask_d   EQU s3

        LOAD bit_mask_d, $80       ;Start with MSB.
        LOAD remainder, $00        ;Clear remainder.

        div_loop:
        CALL flash_LEDs            ;Flash whilst code executing.
        TEST s0, bit_mask_d        ;Set Carry for test bit=1.
        SLA remainder              ;Shift CARRY to lsb of the remainder.
        SL0 quotient               ;quotient shifted left.

        COMP remainder, s2         ;Compare remainder to divisor.
        JUMP C, no_sub             ;Divisor greater -> next bit (no sub).
        SUB remainder, s2          ;Remainder greater -> substraction.
        ADD quotient, 01           ;Add one to quotient.

        no_sub:
        SR0 bit_mask_d             ;shift bit_mask_d to next bit position.
        JUMP NZ, div_loop          ;Back to div_loop if not zero.

        OUT  quotient, char21      ;Calculation output to char21 LED.
        JUMP     FINISH            ;Go to sequence finish.
```

```
multiplication:
        ;Register Declaration:
        ;multiplicand- s0
        ;multiplier - s2
        bit_mask     EQU s5
        result_msb   EQU s6                ;MSB result.
        result_lsb   EQU s7                ;LSB result.

        LOAD bit_mask, $01                 ;Start at LSB.
        LOAD result_msb, $00               ;Clear.
        LOAD result_lsb, $00               ;Clear.

        ; Looping through all bits in multiplier
        mult_loop:
        CALL flash_LEDs                    ;Flash LEDs during calcualtion.
        TEST s2, bit_mask                  ;Check if bit is high.

        JUMP Z, no_add                     ;Bit not set -> skip add.
        ADD result_msb, s0                 ;Occurs in MSB.
        no_add:
        SRA result_msb                      ;Right shift MSB, CARRY to bit 7,
                                            ;LSB in CARRY.
        SRA result_lsb                      ;Right shift LSB. LSB from result_msb
                                            ;into bit 7.
        SL0 bit_mask                        ;Check next bit in multiplier by
                                            ;shifting bit_mask.
        JUMP NZ, mult_loop                  ;if not zero go back to mult_loop.
        OUT result_msb, char21              ; MSB in top display.
        OUT result_lsb, char43              ; LSB in bottom display.
        JUMP     FINISH                     ;Go to sequence finish.
```

II. Simulations

This part of the report demonstrates the functionalities of the programme described in the previous part of this report. The simulation was realised using pBlaze allowing to simulate inputs (8 bit binary: values and buttons) and outputs (7 segment display for numerical values and LEDs for programme selection and processing activities).

Value and Operation Input:

This section is common to all calculations. The values are inputted as 8-bit sequences on the SWITCHES_port then displayed onto char43. In the example below (Figure II.1), “00000100” ($04) has been input and displayed. The process is repeated for the second value “000000010” ($02) (Figure II.2). The first value is still displayed and will then be replaced by the second value on the display. The operation can then be selected on BUTTONS_port (here addition button 1 - $02 or “00000010”) and displayed on LEDs_port (Figure II.3). Figure II.4 shows the numerical and operator stored in the registers: s0 - first value, s1 - operator and s2 - second value.

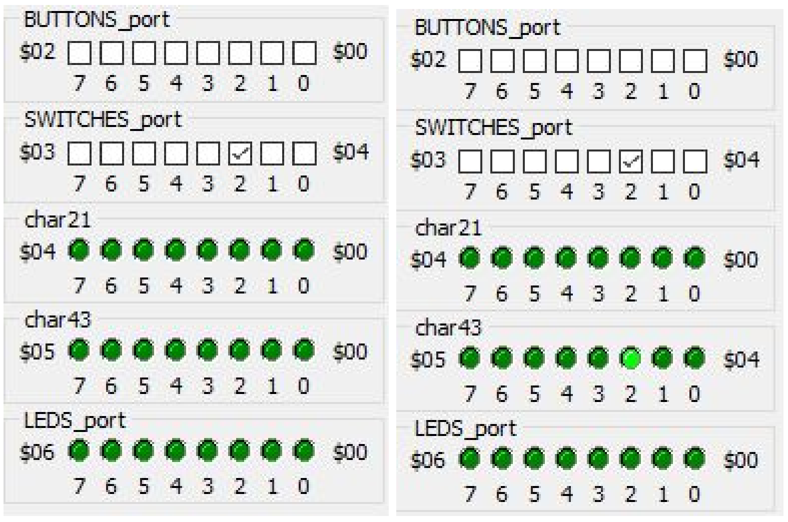
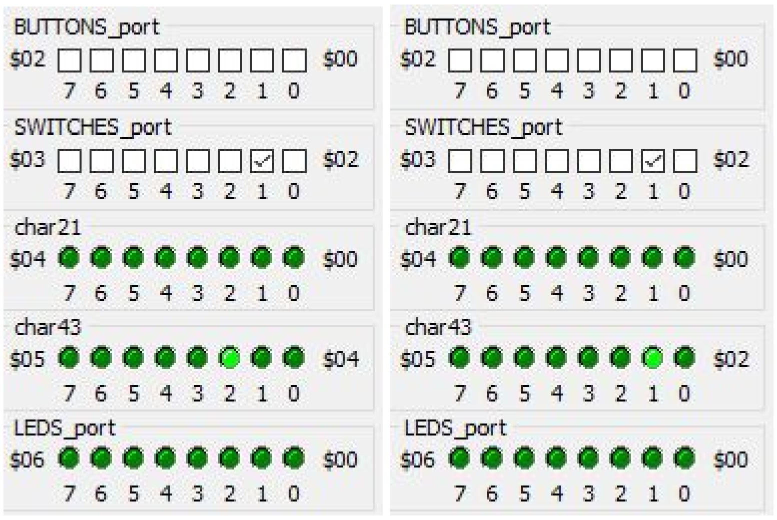
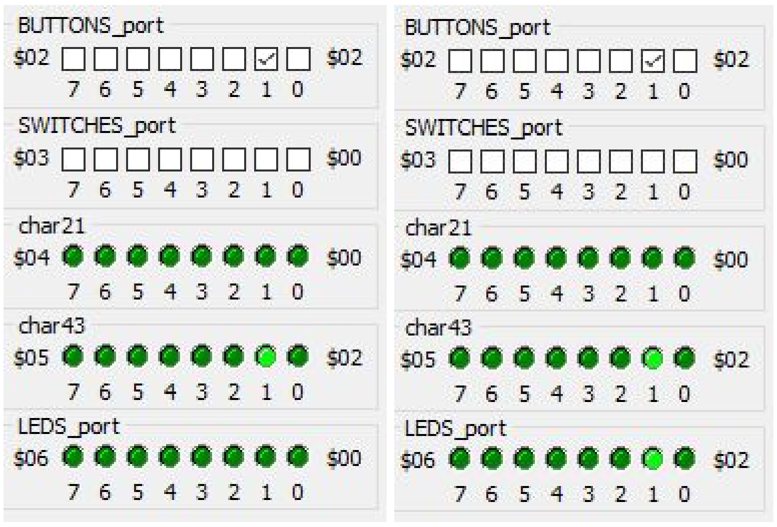


Fig II.1 First value input.



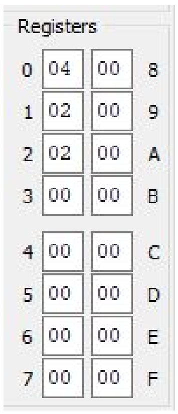
```
BUTTONS_port                          BUTTONS_port
$02 ☐ ☐ ☐ ☐ ☐ ☐ ☐ ☐ $00               $02 ☐ ☐ ☐ ☐ ☐ ☐ ☐ ☐ $00
    7 6 5 4 3 2 1 0                       7 6 5 4 3 2 1 0
SWITCHES_port                         SWITCHES_port
$03 ☐ ☐ ☐ ☐ ☐ ☐ ☑ ☐ $02               $03 ☐ ☐ ☐ ☐ ☐ ☐ ☑ ☐ $02
    7 6 5 4 3 2 1 0                       7 6 5 4 3 2 1 0
char21                                char21
$04 ● ● ● ● ● ● ● ● $00               $04 ● ● ● ● ● ● ● ● $00
    7 6 5 4 3 2 1 0                       7 6 5 4 3 2 1 0
char43                                char43
$05 ● ● ● ● ● ○ ● ● $04               $05 ● ● ● ● ● ● ○ ● $02
    7 6 5 4 3 2 1 0                       7 6 5 4 3 2 1 0
LEDS_port                             LEDS_port
$06 ● ● ● ● ● ● ● ● $00               $06 ● ● ● ● ● ● ● ● $00
    7 6 5 4 3 2 1 0                       7 6 5 4 3 2 1 0
```

Fig II.2 Second value input



```
BUTTONS_port                          BUTTONS_port
$02 ☐ ☐ ☐ ☐ ☐ ☐ ☑ ☐ $02               $02 ☐ ☐ ☐ ☐ ☐ ☐ ☑ ☐ $02
    7 6 5 4 3 2 1 0                       7 6 5 4 3 2 1 0
SWITCHES_port                         SWITCHES_port
$03 ☐ ☐ ☐ ☐ ☐ ☐ ☐ ☐ $00               $03 ☐ ☐ ☐ ☐ ☐ ☐ ☐ ☐ $00
    7 6 5 4 3 2 1 0                       7 6 5 4 3 2 1 0
char21                                char21
$04 ● ● ● ● ● ● ● ● $00               $04 ● ● ● ● ● ● ● ● $00
    7 6 5 4 3 2 1 0                       7 6 5 4 3 2 1 0
char43                                char43
$05 ● ● ● ● ● ● ○ ● $02               $05 ● ● ● ● ● ● ○ ● $02
    7 6 5 4 3 2 1 0                       7 6 5 4 3 2 1 0
LEDS_port                             LEDS_port
$06 ● ● ● ● ● ● ● ● $00               $06 ● ● ● ● ● ● ○ ● $02
    7 6 5 4 3 2 1 0                       7 6 5 4 3 2 1 0
```

Fig II.3 Operator selection



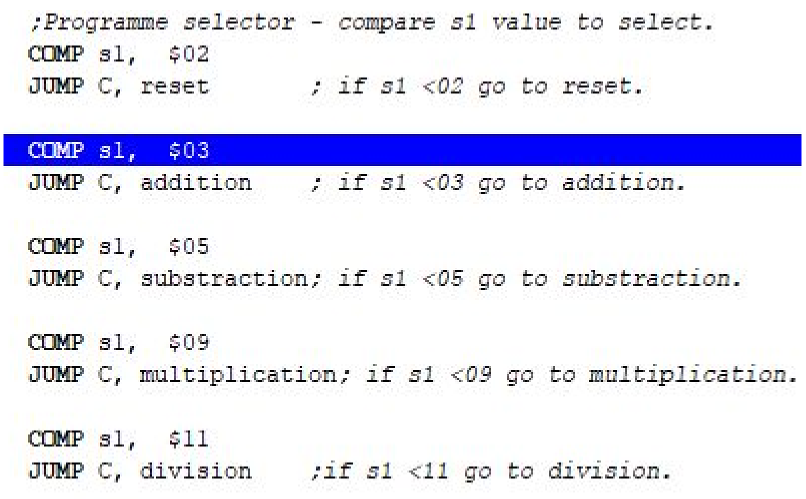
Registers

0	04	00	8
1	02	00	9
2	02	00	A
3	00	00	B
4	00	00	C
5	00	00	D
6	00	00	E
7	00	00	F

Fig II.4 Registers

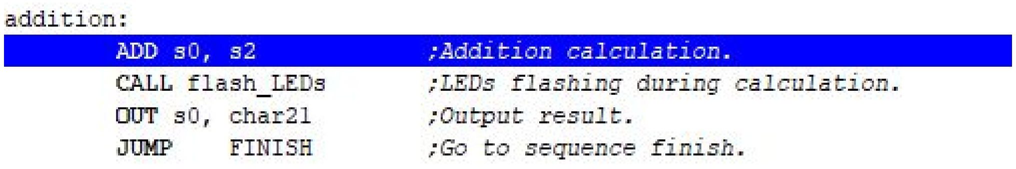
Addition:

Following from the previous inputted values and operator, figure II.5 shows the programme comparing the value stored in s1 to the various operation routine indexes. The programme then jumps to the addition routine (Figure II.6) and realises the operation whilst flashing the LEDs (Figure II.7).



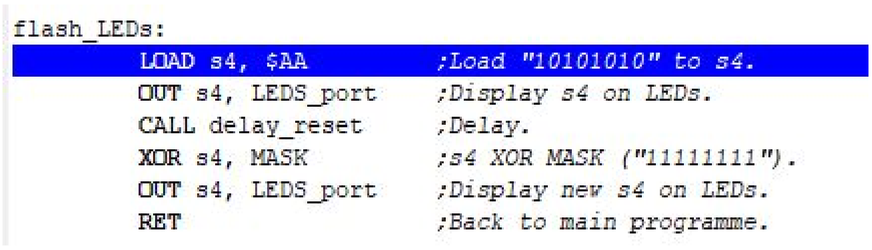
```
;Programme selector - compare s1 value to select.
COMP s1,  $02
JUMP C, reset       ; if s1 <02 go to reset.

COMP s1,  $03
JUMP C, addition    ; if s1 <03 go to addition.

COMP s1,  $05
JUMP C, substraction; if s1 <05 go to substraction.

COMP s1,  $09
JUMP C, multiplication; if s1 <09 go to multiplication.

COMP s1,  $11
JUMP C, division    ;if s1 <11 go to division.
```

Figure II.5 Comparing s1 to operations indexes

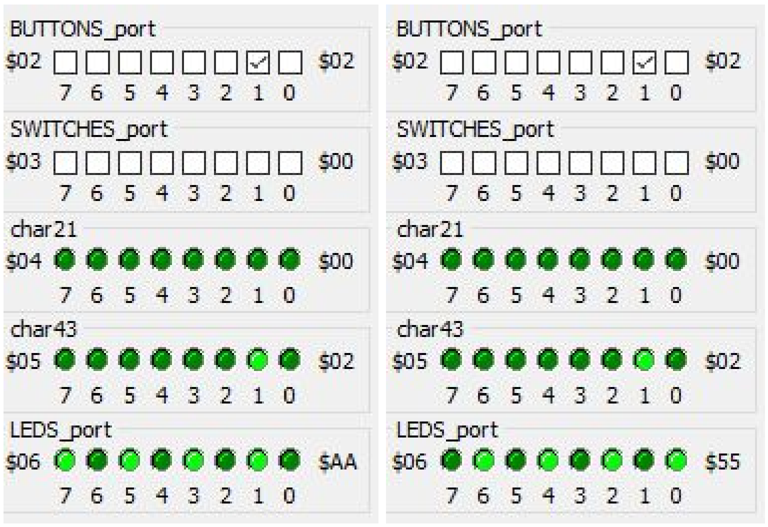


```
addition:
        ADD s0, s2              ;Addition calculation.
        CALL flash_LEDs         ;LEDs flashing during calculation.
        OUT s0, char21          ;Output result.
        JUMP    FINISH          ;Go to sequence finish.
```

Figure II.6 Addition routine.



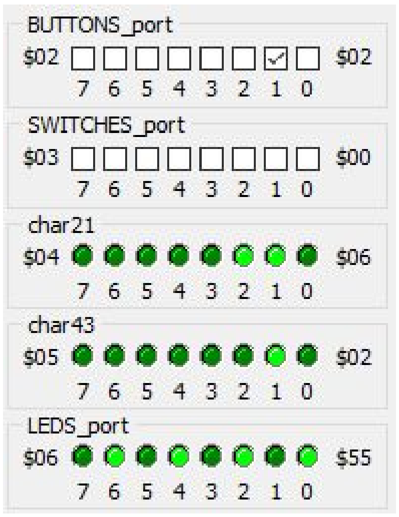
```
flash_LEDs:
          LOAD s4, $AA          ;Load "10101010" to s4.
          OUT s4, LEDS_port     ;Display s4 on LEDs.
          CALL delay_reset      ;Delay.
          XOR s4, MASK          ;s4 XOR MASK ("11111111").
          OUT s4, LEDS_port     ;Display new s4 on LEDs.
          RET                   ;Back to main programme.
```



```
BUTTONS_port                     BUTTONS_port
$02 ☐ ☐ ☐ ☐ ☐ ☐ ☑ ☐ $02          $02 ☐ ☐ ☐ ☐ ☐ ☐ ☑ ☐ $02
    7 6 5 4 3 2 1 0                  7 6 5 4 3 2 1 0
SWITCHES_port                    SWITCHES_port
$03 ☐ ☐ ☐ ☐ ☐ ☐ ☐ ☐ $00          $03 ☐ ☐ ☐ ☐ ☐ ☐ ☐ ☐ $00
    7 6 5 4 3 2 1 0                  7 6 5 4 3 2 1 0
char21                           char21
$04 ● ● ● ● ● ● ● ● $00          $04 ● ● ● ● ● ● ● ● $00
    7 6 5 4 3 2 1 0                  7 6 5 4 3 2 1 0
char43                           char43
$05 ● ● ● ● ● ● ● ● $02          $05 ● ● ● ● ● ● ● ● $02
    7 6 5 4 3 2 1 0                  7 6 5 4 3 2 1 0
LEDS_port                        LEDS_port
$06 ● ● ● ● ● ● ● ● $AA          $06 ● ● ● ● ● ● ● ● $55
    7 6 5 4 3 2 1 0                  7 6 5 4 3 2 1 0
```

Figure II.7 Flash LEDs routine.

The result is then displayed on char21 (figure II.8) with the programme jumping to the "FINISH" routine whereLEDs flashing 4 times to signify the end of the calculation and the programme jump to "reset" (detailed later) for the next calculation.



```
BUTTONS_port
$02 ☐ ☐ ☐ ☐ ☐ ☐ ☑ ☐ $02
    7 6 5 4 3 2 1 0
SWITCHES_port
$03 ☐ ☐ ☐ ☐ ☐ ☐ ☐ ☐ $00
    7 6 5 4 3 2 1 0
char21
$04 ● ● ● ● ● ● ● ● $06
    7 6 5 4 3 2 1 0
char43
$05 ● ● ● ● ● ● ● ● $02
    7 6 5 4 3 2 1 0
LEDS_port
$06 ● ● ● ● ● ● ● ● $55
    7 6 5 4 3 2 1 0
```

Figure II.8 Addition result output (4 + 2 = 6).

Substraction:

Substraction works in the same manner as the addition, the only difference being the button selected as shown in figure II.9.

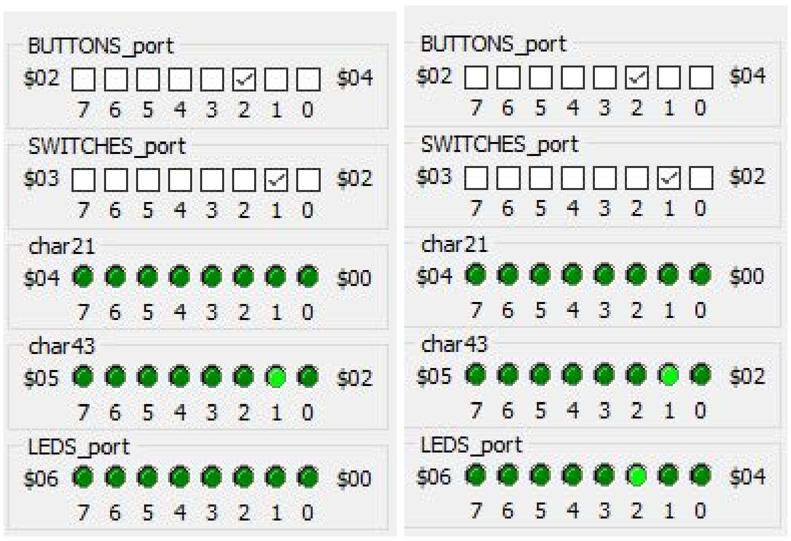


Figure II.9 Substraction operator selection and display.

The programme proceeds on comparing the value in the register to select the corresponding routine. The result of the substraction can be observed in figure II.10.

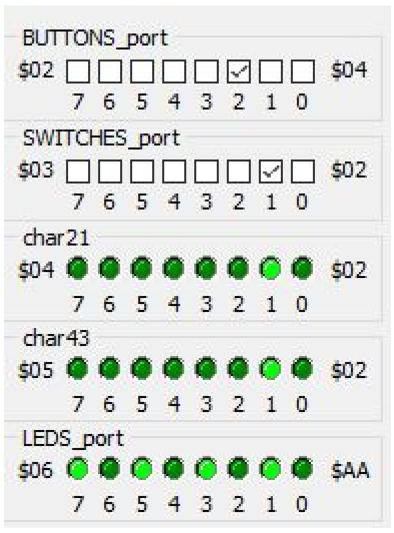
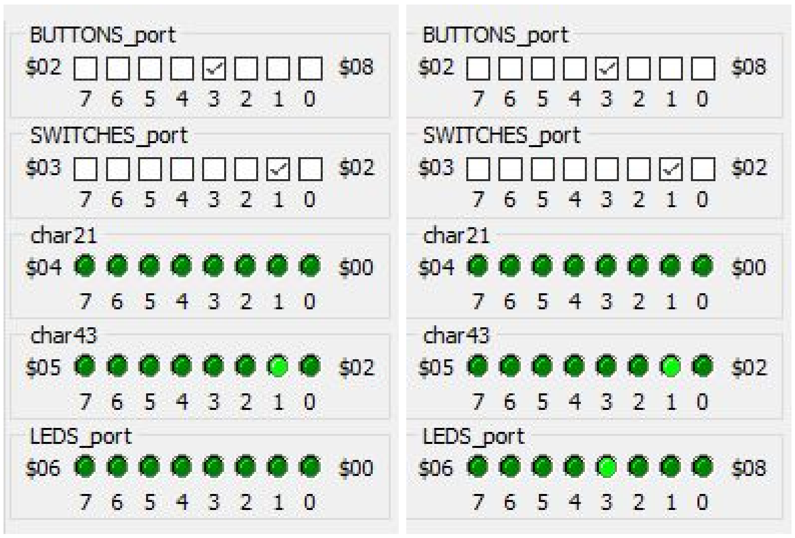


Figure II.10 Substraction operation output (4 - 2 = 2).

Multiplication:

Numerical values and operator are inputted the same way as previously here: 4, 2 and “00001000” ($08) - button 3 for multiplication as shown on figure II.11.

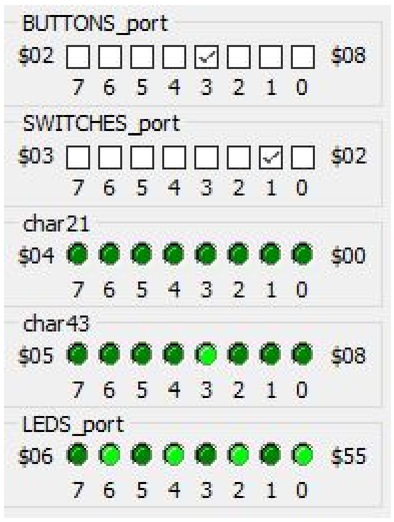


```
BUTTONS_port
$02 ☐ ☐ ☐ ☐ ☑ ☐ ☐ ☐ $08
    7 6 5 4 3 2 1 0
SWITCHES_port
$03 ☐ ☐ ☐ ☐ ☐ ☐ ☑ ☐ $02
    7 6 5 4 3 2 1 0
char21
$04 ● ● ● ● ● ● ● ● $00
    7 6 5 4 3 2 1 0
char43
$05 ● ● ● ● ● ● ● ● $02
    7 6 5 4 3 2 1 0
LEDS_port
$06 ● ● ● ● ● ● ● ● $00
    7 6 5 4 3 2 1 0
```

```
BUTTONS_port
$02 ☐ ☐ ☐ ☐ ☑ ☐ ☐ ☐ $08
    7 6 5 4 3 2 1 0
SWITCHES_port
$03 ☐ ☐ ☐ ☐ ☐ ☐ ☑ ☐ $02
    7 6 5 4 3 2 1 0
char21
$04 ● ● ● ● ● ● ● ● $00
    7 6 5 4 3 2 1 0
char43
$05 ● ● ● ● ● ● ● ● $02
    7 6 5 4 3 2 1 0
LEDS_port
$06 ● ● ● ● ● ● ● ● $08
    7 6 5 4 3 2 1 0
```

Figure II.11 Multiplication operation selection and display.

The programme will then select the multiplication subroutine and proceed with the calculation and return a value (figure II.12). It is worth mentioning the multiplication routine requires more processing than substraction and addition.



```
BUTTONS_port
$02 ☐ ☐ ☐ ☐ ☑ ☐ ☐ ☐ $08
    7 6 5 4 3 2 1 0
SWITCHES_port
$03 ☐ ☐ ☐ ☐ ☐ ☐ ☑ ☐ $02
    7 6 5 4 3 2 1 0
char21
$04 ● ● ● ● ● ● ● ● $00
    7 6 5 4 3 2 1 0
char43
$05 ● ● ● ● ● ● ● ● $08
    7 6 5 4 3 2 1 0
LEDS_port
$06 ● ● ● ● ● ● ● ● $55
    7 6 5 4 3 2 1 0
```

Figure II.12 Multiplication operation output (4 x 2 = 8)

Division:

The division works in a similar fashion as the multiplication. It requires more processing to obtain a value. Figure II.13 demonstrates the operator selection (button 4) and output on the display

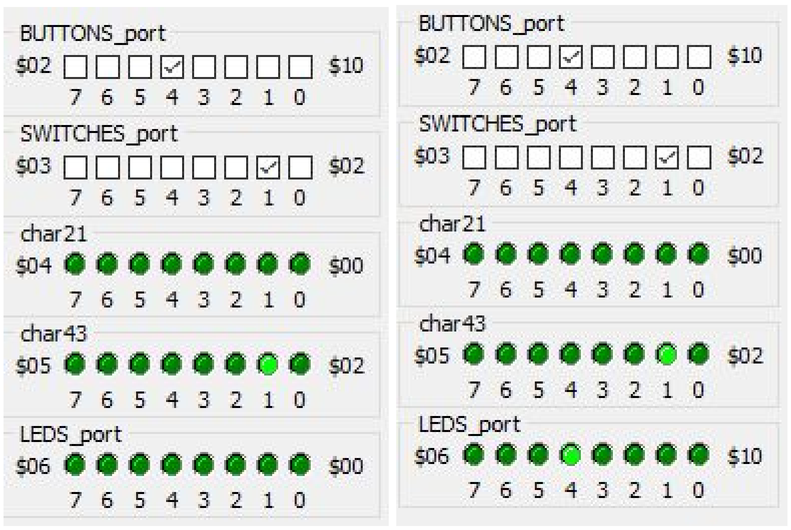


Figure II.13 Division operation selection and display.

The programme will then select the division subroutine and proceed with the calculation and return a value (figure II.14).

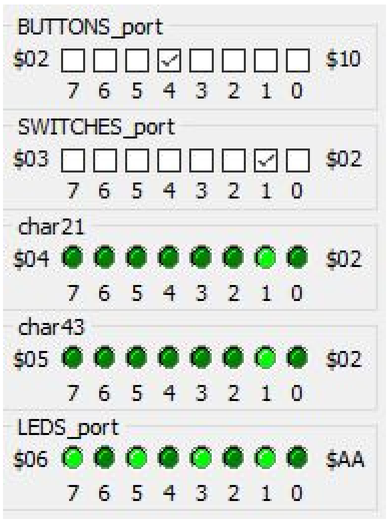
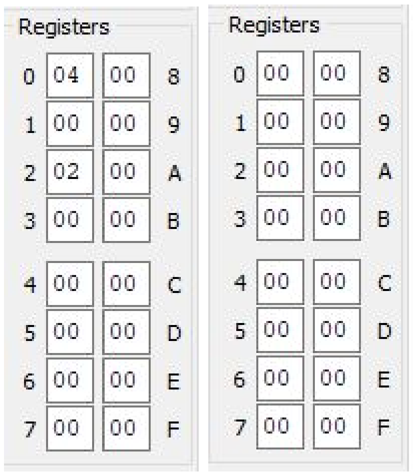


Figure II.14 Division operation output (4 / 2 = 2)

Reset:

Reset can be selected using button 0 ($01) but also executes at the end of every calculation. It clears all the registers (Figure II.14) and display outputs (Figure II.15).



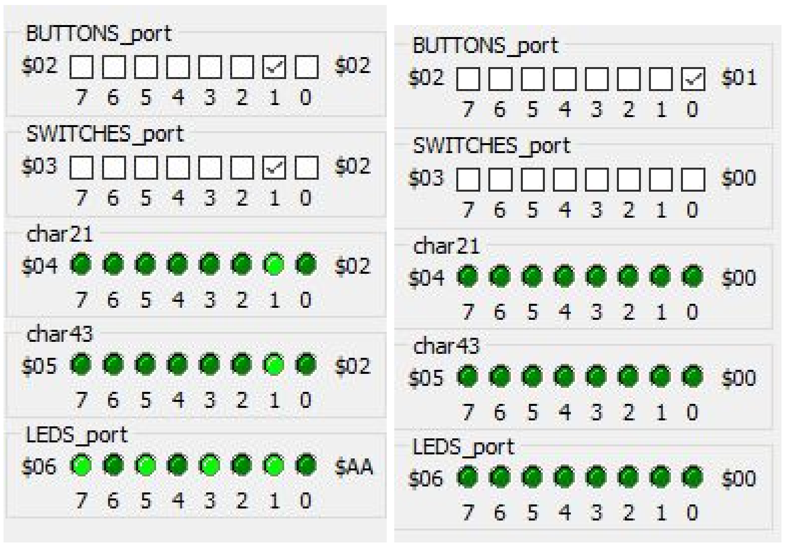
Registers

0	04	00	8
1	00	00	9
2	02	00	A
3	00	00	B
4	00	00	C
5	00	00	D
6	00	00	E
7	00	00	F

Registers

0	00	00	8
1	00	00	9
2	00	00	A
3	00	00	B
4	00	00	C
5	00	00	D
6	00	00	E
7	00	00	F

Figure II.14 Registers before/after.



BUTTONS_port									
$02	☐	☐	☐	☐	☐	☐	✓	☐	$02
	7	6	5	4	3	2	1	0	
SWITCHES_port									
$03	☐	☐	☐	☐	☐	☐	✓	☐	$02
	7	6	5	4	3	2	1	0	
char21									
$04	●	●	●	●	●	●	🟢	●	$02
	7	6	5	4	3	2	1	0	
char43									
$05	●	●	●	●	●	●	🟢	●	$02
	7	6	5	4	3	2	1	0	
LEDS_port									
$06	🟢	●	🟢	●	🟢	●	🟢	●	$AA
	7	6	5	4	3	2	1	0	

BUTTONS_port									
$02	☐	☐	☐	☐	☐	☐	☐	✓	$01
	7	6	5	4	3	2	1	0	
SWITCHES_port									
$03	☐	☐	☐	☐	☐	☐	☐	☐	$00
	7	6	5	4	3	2	1	0	
char21									
$04	●	●	●	●	●	●	●	●	$00
	7	6	5	4	3	2	1	0	
char43									
$05	●	●	●	●	●	●	●	●	$00
	7	6	5	4	3	2	1	0	
LEDS_port									
$06	●	●	●	●	●	●	●	●	$00
	7	6	5	4	3	2	1	0	

Figure II.15 Display before/after.

III. Discussion: decisions, problems, errors and assessment.

This section serves as a reflection on the work undertaken in this assignment. It summarises some of the decisions made, problems and errors encountered and a personal assessment of work realised.

First, it was assumed the design would only need to perform simple calculations. The code performs on operation between two numerical values and then resets. If the output was required to perform another operation, the feedback to the input would be managed by the larger FPGA design. It was also decided to display the result of the multiplication on two outputs one for each 8 bit result sequence (MSB, LSB).

Secondly, pBlaze is sadly not a maintained software which makes it obsolete. Moreover the testing graphical interface is very rudimentary but a useful feature for rapid prototyping especially in complement of the step by step programme processing. pBlaze uses its own derived assembly language but only offers the ROM file for integration in the FPGA. The psm programme file is not compatible with KCPSM assembler, because of the derived assembly language, therefore it is not possible to generate the “controller” vhd file. The code would then have to be reimplemented in assembly to be compiled with an appropriate assembler and then integrated into a VHDL project.

Finally, the design provided is very basic and it is believed more work should be done to develop a more complex integrated solution. Due to time restrictions on the project the solution selected was the most simplistic and straightforward to implement.

Conclusion.

Picoblaze is a very light and flexible package microcontroller core package which can easily be integrated in FPGA projects requiring a microcontroller which has a fast execution, response and allows for future upgrades. This report gave a short explanation of the basics of a Reverse Polish Notation calculator. Using pBlaze a simple RPN calculator was created and tested using the provided graphical interface. A report of the testing is provided demonstrating the basics of functions: addition, substraction, multiplication and division as well as accessory operations such as: resetting and flashing LEDs. It is believed more work needs to be conducted to develop a more integrated solution to the calculator problem. Although the design is functional, it can only perform one operation and then reset if used on its own. If integrated to a larger FPGA design accessory functions can be added to remedy this and reload previous calculation results to the input of the picoblaze.

References.

Stone, A. (2018). *Reverse Polish Notation*. [online] www-stone.ch.cam.ac.uk. Available at: http://www-stone.ch.cam.ac.uk/documentation/rrf/rpn.html [Accessed 10 May 2020].

Xilinx and Inc. (2011). *PicoBlaze 8-bit Embedded Microcontroller User Guide for Extended Spartan ® -3 and Virtex ® - 5 FPGAs Introducing PicoBlaze for Spartan-6, Virtex-6, and 7 Series FPGAs*. [online] Available at: https://www.xilinx.com/support/documentation/ip_documentation/ug129.pdf [Accessed 10 May 2020].