TABLE OF CONTENTS

1 INTRODUCTION

This document is a specification of a simple hardware calculator, which is to be implemented on an FPGA board. The calculator reads input from a keyboard attached to the board via PS/2 interface. User input, results, and error messages are displayed at a monitor via VGA interface. On demand, last 50 calculations (expressions and results) are transferred to a PC terminal over RS-232. The calculator works only with signed integers in the range $[-2^{32}, 2^{32}-1]$. You can see a detailed list of requirements in the next section.

The calculator can compute the following arithmetic operations: *addition, subtraction, multiplication,* and *division.* The user's input is limited to digits, operator signs, spaces, backspaces, and hitting enter. A user cannot use parentheses to influence the order of expression evaluation; however multiplication and division have higher priority than addition and subtraction. The fact, that parenthesis are not allowed simplifies the HW design significantly. We can avoid using stacks and get along with temporal registers. If parentheses were allowed, we would need to opt for somewhat more sophisticated solution such as reordering expressions into *reversed Polish notation.* However, given the requirements, we do not need to reorder the user input.

This document is a part of the assignments for Hardware Modeling course (no. 182129), given at Vienna University of Technology.

2 REQUIREMENTS

In follow there is a list of requests that calculator should be capable to do it.

Req 1: Calculator should be designed on a FPGA board;

Req 2: Data input should be done by keyboard through PS/2 port;

Req 3: Data output should be done by at display in one line. Communication has to be through VGA port;

Req 4 : Calculator has to store in buffer memory last 50 arithmetic expressions and results.

Req 5 : Buffer memory content should be send to PC by RS232 port.

Req 6: Calculator should calculate arithmetic expressions which comply to the following format:

```
EXPRESSION     = OPERAND {OPERATOR, OPERAND};
OPERATOR       = "+" | "-" | "*" | "/";
OPERAND        = ["-"], UNSIGNED;
UNSIGNED       = DIGIT, {DIGIT};
DIGIT          = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9";
```

Req 7 : Calculator work only with signed integer numbers.

Req 8: Range of date it's integer from (-2^{31}) to $(2^{31}-1)$.

Req 9: Display can show up to 70 characters in one line.

Req 10 : Space character should be shown on screen.

Req 11 : Hitting 'BACKSPACE' should be delete last character.

Req 12 : Hitting 'ENTER' key should start the computation. When the computation is finished and result displayed, the calculator waits for a next input.

Req 13 : The precedence in expression evaluation is given by two classes of operators priority: high priority class contains * and / and low priority class contains + and – operators.

3 HIGH-LEVEL DESIGN DESCRIPTION

3.1 OPERATION

The operation of the FPGA calculator is schematically depicted in Figure 1. The operation starts in state **A**, reading the input from a user until the user hits 'ENTER'. While reading, the characters are being displayed at a VGA monitor. After a user commits the expression by hitting 'ENTER', we check whether the arithmetical expression is syntactically correct. If there is an error, we transit to the state **B**. The activity in state **B** is to display appropriate message on a VGA monitor and reset the system, so that it is ready to accept new input, the calculator then immediately proceeds back to state **A**. If the expression is correct, the calculation starts in state **C**. When the result is ready, the calculator logs the computation in a ring memory (state **D**) and then proceeds to displaying the result (state **E**). Afterward, the system is reset to its initial state **A**.

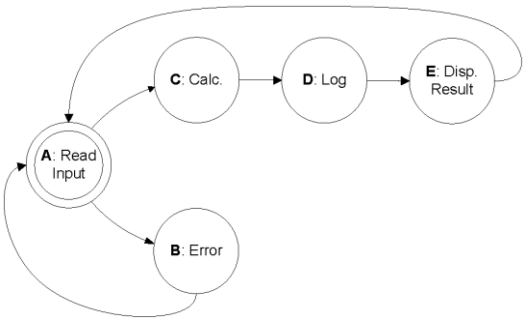

Figure 1: High-level operation of the calculator. Each of the state represents an activity, which cannot be, in our design, executed in parallel with the rest of the activities.

During logging (state **D**), the serial communication via RS-232 must be disabled. Also if the serial communication is on the way, while transiting from **C** to **D**, the transition is delayed, until the transfer completes. In other states, RS232 transmission may run in parallel.

3.2 STRUCTURE

FPGA calculator contains four main units (Figure 2). The green block represent an input unit to the FPGA calculator, the yellow block is a unit that executes calculations, the red block displays typed numbers, operators, and results, and the blue unit is used to log arithmetic expressions together with their results to the memory. The arrows represent data flow between modules. Control signals are not depicted in the figure.

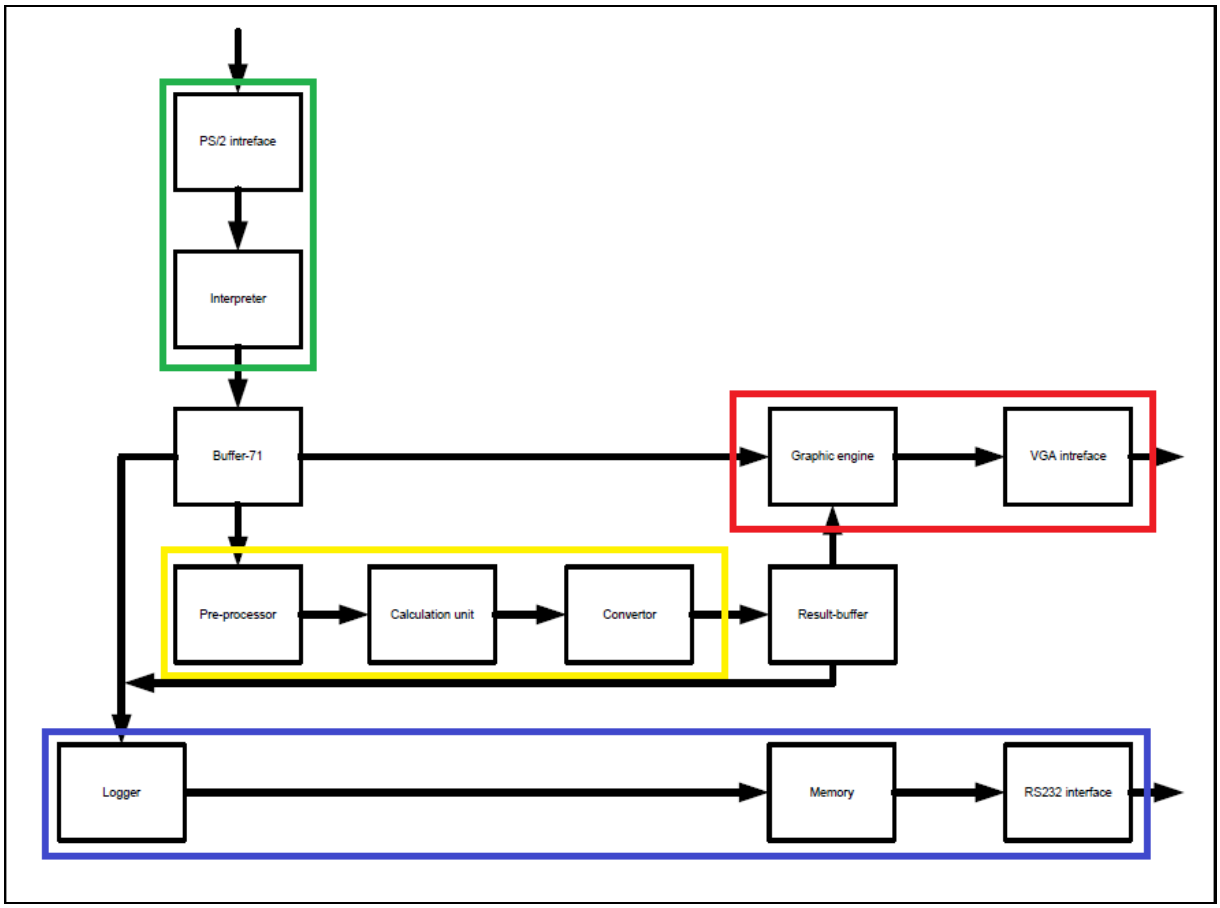

Figure 2: Main calculator modules

Each task is presented as a separate module. Our design contains twelve modules the role of which is described in the following:

- **PS/2 interface** reads scancodes (user data) generated from a keyboard and deliver this data to the *Interpreter*.
- **Interpreter**'s task is to interpret the scancodes, store the user-defined arithmetical expressions in the memory (Buffer-71) and keep the memory consistent according to the semantics of the scancodes. If, for example, a scancode for 'BACKSPACE' arrives, Interpreter needs to update the buffer accordingly. When scancode for 'ENTER' arrives, the Interpreter checks, whether the stored expression is syntactically correct and either triggers the computation or raises an error. Throughout its operation, Interpreter informs *Graphic engine* whenever *Buffer-71* memory has changed and therefore display update is due.
- **Buffer-71** stores the user-defined arithmetic expression which (based on req. 9) is limited to a length of 70 characters. The 71th byte is used for storing the code of 'ENTER' button.
- **Pre-processor** reads the content of Buffer-71 in a loop and produces in each cycle a pair of data: *<operand, operator>*, where *operand* is 32bit signed integer and *operator* represents a code of the operation to be computed. This data pair is fed to the c*alculation unit.* It is the pre-processor, who converts the ASCII representation of operands to the 4 bytes-long two's complement representation.
- **Calculation unit** operates in cycles. In each cycle it reads the data-pair *<operand, operator>* from the pre-processor unit and computes the intermediate result. When finished with calculation, it sends the result to *Convertor*.

- **Convertor** converts the results from *Calculation unit* into the ASCII representation and stores it in the *Result-buffer*.
- We use the **Result-buffer** for storing the calculated and converted result of the arithmetic expression.
- **Graphic engine** is a special-purpose interface between the system and the VGA controller. The graphic engine reads a given portion of a memory (defined by start address and end address), interprets the content as characters and displays them at a VGA monitor in one of three modes: expression mode, result mode, and error mode.
- **VGA** is the provided interface to the VGA port.
- **Logger**'s task is to read both the Buffer-71 and Result-buffer when the computation is finished and store them in the ring buffer represented by the *Memory* module.
- **Memory** is a ring buffer that keeps the last 50 expressions and their results.
- **RS232** transfers on demand the content of the ring-buffger from the *Memory* module to a PC via RS232 interface.

3.3 EXTERNAL INTERFACES

Calculator designed at FPGA board has to communicate with three outside units: keyboard, monitor and PC. Keyboard is used as input while monitor and PC as output unit.

Physical interface with keyboard: Keyboard is communicating with FPGA calculator through PS/2 connector. There must be defined two bidirectional one bit signals (*ps2_clk,* and *ps2_data*).

Interface behavior with keyboard: External PS/2 interface behavior depends from *ps2_clk, ps2_data* which are input signal to our *PS/2 module*. *ps2_clk* is clock signal while *ps2_data* is data signal. The basic function is to sample *ps2_data* signal after each change of *ps2_clk* clock signal.

Physical interface with monitor: Monitor is connected with FPGA through VGA connector. Communication is established through seven signals(*vga_clk, vga_res_n, vsync_n, hsync_n, r, g,* and *b*). *vga_clk* and *vga_res_n* are one bit input signals used to establish VGA timing of clock frequency and to reset VGA timing generator(*vga_res_n* is low active, not synchronized signal). The other two signal (*vsync_n* and *hsync_n*) are also one bit output signals to activate horizontal and vertical synchronization. The last three signals (*r,g* and *b*) are red, green and blue output.

Interface behavior with monitor: External VGA interface has to provide an image display on the screen. The two signals (*vsync_n* and *hsync_n*) are used to control the timing of the scan rate, where *vsync_n* is used for row while *hsync_n* for entire screen. The three color signals are use to control a color of the pixel at a location on the screen.

Physical interface with PC: PC is connected with FGPA through RS232. There are fourth signals defined *trans_data, req_to_send, clear_to_send,* and *carrier_det*. The first signal (*trans_data*) is one bit bidirection signal, while the two other ones (*req_to_send* and *clear_to_send*) are output one bit signal. *carrier_det* is one bit input signal.

Interface behavior with PC: External RS232 interface has to send data to the PC. Firstly *req_to_send* signal is asserted form FPGA to initiate transmission and then *trans_data* send data. If transmission is in reverse direction then *clear_to_send* is requesting permission for data receiving. This signal is accepted by *carrier_det.*

3.4 INTERFACES OF MODULES

3.4.1 MODULE 1 – PS/2

PS/2 module has three signals in interface with *Interpretor*. *sys_clk* and *sys_res_n* are one bit input signals. *sys_clk* is system clock signal while *sys_res_n* is system reset signal and is low active. Output signal is *new_data* used to signalize the availability of a new scancode. Scancode is send through 8 bits width *data* signal.

3.4.2 MODULE 2 – INTERPRETER

INPUT: scancode signals from from PS/2 interface *new_data/1* and *data/8*

OUTPUT:	*data/8*	Interface to the buffer-71
	address/?	Interface to the buffer-71
	rw/1	Interface to the buffer-71
	expr_ready/1	Raised when user typed 'ENTER' and expression is syntactically correct
	error/1	Raised when user typed 'ENTER' and expression is syntactically **incorrect**

3.4.3 MODULE 3 – BUFFER-71

The Buffer-71 is a simple memory with standard interface: *data, address, rw* (read/write), and possibly *cs* (chips select).

3.4.4 MODULE 4 – PRE-POCESSOR

INPUT:	*data/8*	Interface to the buffer-71
	calc_ready/1	Input from calculation unit. Signals that calculation unit is ready to accept new <*operand, operator*> pair
OUTPUT:	operand/4	The value of a next operand
	operator/3	The code of the next operator
	op_ready/8	Set high if the next operand, operator pair is ready
	add/?	Interface to the buffer-71
	rw/1	Interface to the buffer-71

3.4.5 MODULE 5 – CALCULATION UNIT

INPUT:	operand/32	The value of a next operand
	operator/3	The code of the next operator
	op_ready/8	Set high if the next <*operand, operator*> pair is ready
	reset/1	Resets the calculation unit to its initial state
OUTPUT:	calc_ready/1	Signals that calculation unit is ready to accept new <*operand, operator*> pair
	result /32	The result leaves the unit through these signals
	result_ready/1	Raised high, when the calculation unit has the result ready

3.4.6 MODULE 6 –CONVERTOR

INPUT:	result /32	The result from computation unit
	result_ready/1	Raised high, when the calculation unit has the result ready
OUTPUT:	*busy/1*	Kept up until the converter writes the last byte in the result-buffer.
	res_addr/?	Address lines to buffer-71
	res_data/8	Data lines to ring-buffer memory

3.4.7 MODULE 7 –RESULT BUFFER

The buffer is a simple memory with standard interface: *data, address, rw* (read/write), and possibly *cs* (chips select).

3.4.8 MODULE 8 – GRAPHIC ENGINE

INPUT:	*data/8*	Data line to a selected memory
	start_addr/?	The address where to start reading from a memory
	end_addr/?	The address where to finish reading from a memory
	mode/2	The mode of output (00 - *expression,* 01 - *result,* 10 - *error*)
	update/1	When raised, triggers the operation of graphic engine.
OUPUT:	*busy/1*	Kept up until the graphic engine finishes reading from memory and communicating with the VGA interface.

3.4.9 MODULE 9 – VGA

VGA interface has two input one bit signals (*sys_clk,* and *sys_res_n*) which are system clock signal and system reset signal. When VGA is ready to accept character it activate *free* one bit output signal. Command which should be executed by the controller is received through *data* 8 bits width signal, while the data field of the command through *command_data* 32 bits signal. Both of the signals were input signals.

3.4.10 MODULE 10 – LOGGER

INPUT:	buff_*data/8*	Data lines to buffer-71
	res_*data/8*	Data lines to result-buffer
	result_ready/1	Signals that the logger should save the contents of expression and result buffers to memory.
OUTPUT:	*busy/1*	Kept up until the logger finishes the job
	buff_addr/?	Address lines to buffer-71
	res_addr/?	Address lines to result-buffer
	mem_data/8	Data lines to ring-buffer memory
	mem_addr/?	Address lines to ring-buffer memory

3.4.11 MODULE 11 – MEMORY

Standard interface: *data, address, rw* (read/write), and possibly *cs* (chips select).

3.4.12 MODULE 12 – RS232

RS232 module has *sys_res* one bit input signal generated by FPGA button to initiate transmission, and *read_data* output signals to read data from memory. Signals *addr* and *data* interface this module to the log memory.

4 DETAILED DESIGN DESCRIPTION

In this section, we describe in detail the design of Interpreter and the calculation unit. The rest of modules will be described in the next version of this specification.

4.1 INTERPRETER

Interpreter's task is to interpret the scancodes coming from the PS/2 interface and store the user-defined arithmetical expressions in the memory (Buffer-71) and keep the memory consistent according to the semantics of the scancodes. Second part of the interpreter's responsibility is to check for syntax correctness when a user commits (by hitting 'ENTER') the arithmetic expression. The operation of the interpreter is depicted in Figure 3. Note that invalid characters are ignored and that 'BACKSPACE' is part of the valid characters set.

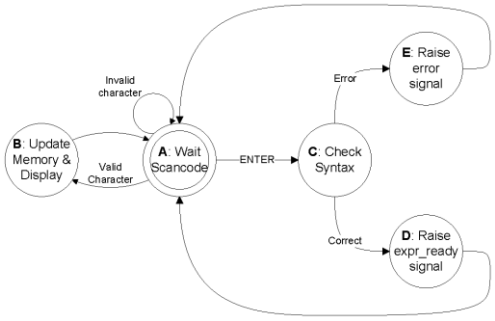

Figure 3: Operation of the Interpreter module

In order to check the syntax of an arithmetic expression, we have translated the grammar specification of valid arithmetic expressions (see **Requirement 6**) into a state machine. The operation of the interpreter, while in the macro-state **C,** is derived from the operation of this state machine in Figure 4. If a transition is not defined in the state machine, the interpreter must signal a syntax error. If, however, the state machine accepts the character string, the interpreter signals that a new expression is ready for evaluation.

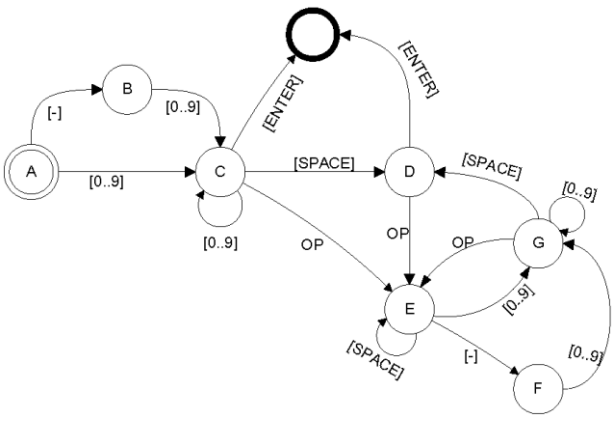

Figure 4: A diagram of a state machine accepting valid arithmetic expressions. This state diagram is part of the C state (Check Syntax) from Figure 3.

4.2 CALCULATION UNIT

The calculation unit operates in cycles. In each cycle, it reads a data-pair *<operand, operator>* from the pre-processor and computes the intermediate result. When finished with calculation, it sends the result to the convertor. As the syntax of arithmetic expressions is rather limited and does not allow for parentheses, we could avoid using for example *reverse polish notation* which is standard yet relatively complex techniques. Our solution does not require reordering of the expression. Below in Figure 5, you can see the scheme of an algorithm, which can evaluate correctly the given format of arithmetic expressions. The algorithm reads the expression from left to right and evaluates it on the fly. The operations *read_operand()* and *read_operator()* represent obtaining the *<operand, operator>* pair from the preprocessor unit. The function *apply(op,A,B),* takes operator as its first argument and operands as its second and third arguments. The function returns the result of applying the operator *op* to the arguments *A*, and *B* as if they were in the form: *"A op B".* For the purpose of the algorithm, we assume that the end of the arithmetic expression is marked by an "operator" 'EQUALS', in other words, the last *<operand, operator>* pair obtained from preprocessor will contain a code for 'EQUALS' as the operator.

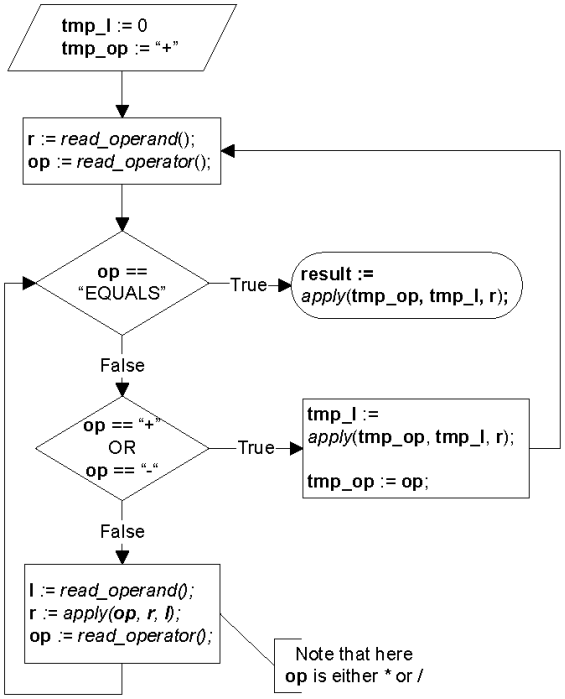

Figure 5: An algorithm for evaluation of arithmetic expressions. The expressions are without parentheses and operators + and – have lower priority than operators / and *.

Note that we initialize the temporal variables such that they are mathematically neutral. In other words, we set the variables in such a way as if we prefixed the arithmetic expression with a string "0 + ".

5 TEST CASES

The following test cases should cover all the testable requirements stated in Section 2.

- TC1: Multiplying one positive and one negative number – covers requirements 2, 12.
- TC2: executing expression that contain adding, subtraction, multiply and division – covers requirement 6 and 13.
- TC3: Writing arithmetical expression longer than 70 character - covers requirements 3, 9.
- TC4: Executing more than 50 arithmetical expressions - covers requirement 4.
- TC5: Pressing FPGA board button for RS232 transmission – covers requirement 5.
- TC6: Executing arithmetical expression that contains 'SPACE' characters within expression – covers requirement 11.
- TC7: Adding one integer and one real number – covers requirement 7.
- TC8: Adding two numbers longer than 2^{31} – covers requirement 8.
- TC9: Executing expression that end with operator – covers requirement 12.