MECHENG 371 – Digital Circuit Design

Simple Calculator project

William Li (916720181)
Kevin Liu (935956894)

1.0 Introduction

This report will discuss the design and implementation of a simple calculator using Verilog HDL on an Altera Cyclone II FPGA. The calculator performs addition, subtraction and multiplication with two operands each with a range of -999 to 999. The calculator also includes simple memory store and recall functions to store numbers for future operations. The user interfaces with the calculator through a 4x4 matrix keypad, push buttons and switches. Operands and answers will be displayed to the user through a suite of eight 7-segment displays.

2.0 System overview

The software is partitioned into five main sections: input, arithmetic calculation, memory, FSM and output. Within each section there are multiple submodules, each module performs a single task and can be used as a standalone function. This is for an easier development process and better code readability. An overall program block diagram can be seen in Section 9.1 of the appendices.

The FPGA given has a 50MHz internal clock, which was stepped down to 500Hz for this application. The clock divider is implemented by incrementing a counter and toggling a second clock signal every 50,000 counts. The speed of 500Hz was chosen as it is both fast for system operations such that the user does no perceive any lag and slow enough for hardware inputs. It is also appropriate as a 50MHz clock will have clock pulses too short for signal to travel through long data paths.

3.0 Input Management

3.1 Matrix Keypad

The matrix keypad is the primary interface the user has with the calculator, it is home to number keys $0-9$, operator keys, equal key, clear key and reset key. The keypad is read through a ring counter that cycles each column and polls each row. It outputs an 8 bit register that describe which key has been pressed. A decoder takes this information and outputs five channels of output signals to distinguish between whether a number, operator, clear, reset or equal key has been pressed.

Each of these signals will pass through a debouncer and a pulser where they are outputted and sent out of the module as a single 1 clock cycle wide pulse containing the decoded information. Each pulse will follow the following format: $abbbb$, where a is the bit that signals a button has been pressed, and the latter four bits contain information about the key. Refer to Figure 1 for a section of the block diagram responsible for input management.

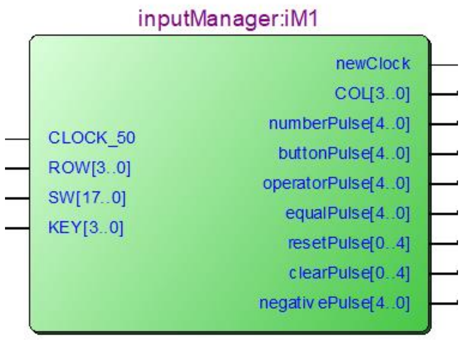


Figure 1 input manager block diagram

3.2 Switches and Buttons

To augment the keypad, the user also has access to a set of push buttons and switches. Push buttons are used for memory management, while a single switch is used to indicate whether the user wishes to enter a negative number.

Memory store/recall/reset switch/pushbutton information is extracted in a standalone module which outputs the relevant information based on whether a button was pressed, and if so, which button. This information is also then deounced, pulsed, and then output in the same form as the matrix keypad.

To enter negative operands, the user can toggle switch 17 whenever an operand is being inputted. The display will also reflect this input by showing a negative sign. The signal from the switch is also processed in a similar manner to the matrix keypad.

4.0 Calculations

The processing of numbers and calculations is done within our arithmetic section, which is broken down into three main components, a BCD to binary converter, an ALU, and a binary to BCD converter. For easier display and shifting of digits in decimal, we have elected to represent all of our numbers as BCD. As a result, to perform arithmetic operations, the number has to be first converted into binary, and after the calculation the answer needs to be converted back to BCD to be displayed to the user.

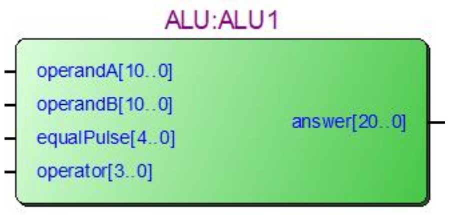


Figure 2 ALU block diagram

To perform a BCD to binary conversion, we multiply each 4-bit binary number in the BCD by its respective place in base 10. This is stored as a signed register in twos compliment form so that we can use Verilog's built in arithmetic operators in the ALU (Figure 2).

For binary to BCD conversion, we considered two different methods: double dabble, and the remainder method. The remainder method continuously takes the remainder of the binary and divides it by ten. As this assignment does not have any performance requirements or memory limitations other than what is available on the DE2 board, we have decided that the simplicity of the remainder method, both to implement and for any future programmers to understand, outweighs the fact that it uses more logic elements.

5.0 Memory

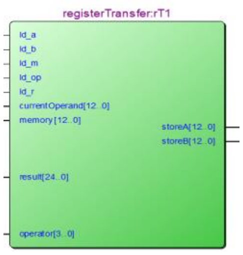
Memory management is responsible for the storage of all variables used in the can be split into two parts: manual memory management from the push buttons, and the management of internal memory.

5.1 Manual Memory Management

Manual memory management utilizes the information extracted from the push buttons, and works by connecting input and outputs for the memory and display register. Based on what the user input, the management system can either store the current output in memory, display the contents of memory on the 7-segment displays to use as an operand, or clear the memory.

The memory store function is used to store a number for future use, and since the calculator only takes in three digit operands, we have decided that if the calculator is displaying more than three digits, it will take the least significant three. While this would not affect either of the operands, this will ultimately limit the ability of users to store results.

5.2 Internal Memory



Internal memory is managed through a series of flags that are controlled by the calculator's finite state machine. The machine outputs a signal for which register to store, reset, and display depending on the state it is in. These flags are fed into a register transfer function (Figure 3) which updates various internal registers based on the current state.

Figure 3 register transfer block diagram

A shift register (Figure 4) was created alongside the register transfer function. The shift register sets the current operand register value depending on certain inputs and flags. If the user presses a number key, it will shift the two less significant digits up and append the new number. The backspace function is also implemented here, by shifting the more significant two bits down and padding the third digit with zero.

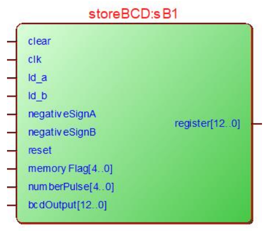
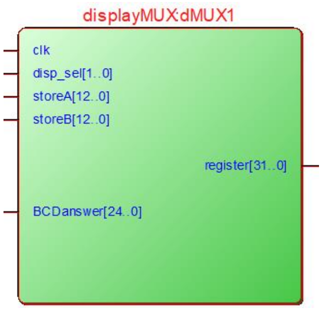


Figure 4 shift register block diagram

6.0 Output



Output of the calculator is displayed on the set of eight 7-segment displays. The number to be displayed is decided by a display selector flag set in the finite state machine. A display multiplexer (Figure 5) takes this selector and dynamically changes the values being stored in the display register.

Figure 5 display multiplexer block diagram

A '–' sign would also be appended onto the eighth (right most) display if the current operand or answer being displayed is negative. In the case that the calculator is in states where it does not explicitly need to display anything, such as when the user has entered an operator, it will retain the last number on the display. This was decided as it may be useful for the user to have a visual record of the first operand even after they have pressed an operator key.

As not all values being outputted will have the same number of digits as there are 7-segment displays, it was decided to pad all displays not being used with zeros. This was deemed to be appropriate as it is intuitive that all leading zeros have no consequence on the number being displayed. A consideration was made to remove leading zeros, however, as the calculator is designed to be purely functional, this change will add extra logic elements without providing any functional benefits.

7.0 FSM

All internal control signals of the calculator are set by the finite state machine (Figure 6). The machine outputs the appropriate flags depending on which state it is in, and state transitions are decided on by user inputs. Refer to Appendices section 9.2 for a diagram of the FSM.

We were presented with 2 main choices for the architecture of the FSM, a Moore type FSM or a Mealy type FSM. Moore has its output based solely on the current state, while Mealy outputs based on both the state and inputs. The former was chosen due to the reliability of transition times offered by the Moore FSM architecture. Although the Moore FSM utilized more elements during implementation, and was slower than the Mealy type it was decided that the reliability of our control unit would be more important than efficiency and speed. This was because state transitions that may occur asynchronously, or too rapidly, could potentially introduce major inaccuracies into the calculator such as affecting the accuracy of register transfer.

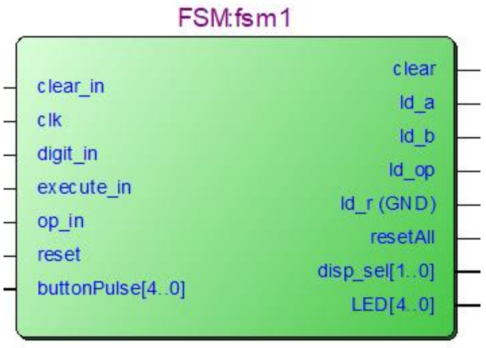


Figure 6 FSM block diagram

Our FSM has 5 states: Reset, OperandA, Operator, OperandB, and Result. On initialisation, the calculator is in reset stage, this resets all internal registers aside from manually stored memory. The transition to OperandA mode is instantaneous and does not require any user input. In OperandA mode, the calculator will continuously load the current number being inputted into a storage register so it can be used later in calculations. When an operator has been pressed, the calculator will store the operator, cease to update OperandA storage, and clear the current number from register. When a number has been pressed again the FSM will transition to its fourth state, where it stores the numbers being entered into an OperandB storage. Finally, when the equals key is pressed the ALU will calculate the answer, and this will be loaded into the display. The reset (A) key will put the calculator into Reset state no matter its current state, while a clear key will clear the current operator but not any other internal registers.

8.0 Conclusions

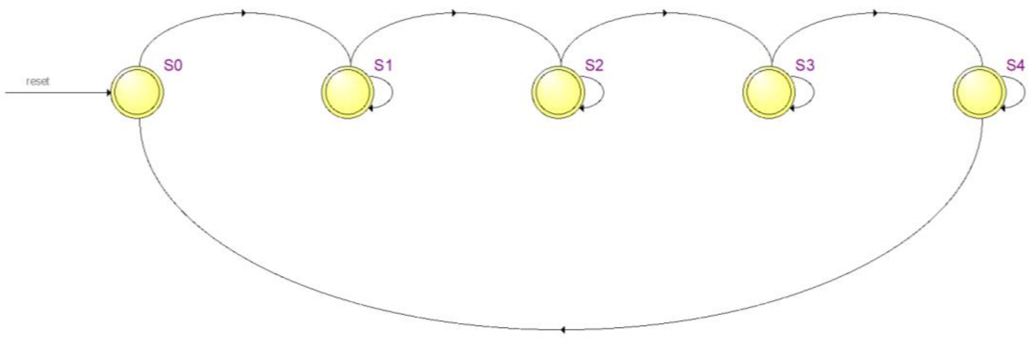
The calculator implemented meets all the required specifications. The calculator is able to meet the following specifications:

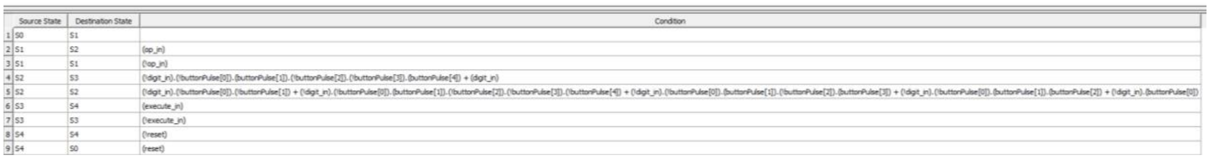
- The correct addition, subtraction and multiplication of two signed decimal numbers between the ranges of 999 to -999
- Memory storage, recall, and clear operations
- A ‘backspace’ function which clears the rightmost digit of an operand
- A correct ‘clear’ implementation which allows users to erase the currently entered number
- A correct reset implementation which resets the calculator back to its initial state.

Some limitations of the system include limited number of digits allowed for operands and a lack of division operator. These are common features of all calculators, however were not implemented as they were out of the scope of the project. To make this program a usable calculator it is recommended anyone carrying on this work implement these features.

9.0 Appendices

9.1 Finite State Machine





	Source State	Destination State	Condition
1	S0	S1	
2	S1	S2	(op_in)
3	S1	S1	(!op_in)
4	S2	S3	(!digit_in).(!buttonPulse[0]).(buttonPulse[1]).(!buttonPulse[2]).(!buttonPulse[3]).(buttonPulse[4]) + (digit_in)
5	S2	S2	(!digit_in).(!buttonPulse[0]).(!buttonPulse[1]) + (!digit_in).(!buttonPulse[0]).(buttonPulse[1]).(!buttonPulse[2]).(!buttonPulse[3]).(!buttonPulse[4]) + (!digit_in).(!buttonPulse[0]).(buttonPulse[1]).(!buttonPulse[2]).(buttonPulse[3]) + (!digit_in).(!buttonPulse[0]).(buttonPulse[1]).(buttonPulse[2]) + (!digit_in).(buttonPulse[0])
6	S3	S4	(execute_in)
7	S3	S3	(!execute_in)
8	S4	S4	(!reset)
9	S4	S0	(reset)

9.2 Block Diagram