17.5.2020

CS 223-4

PROJECT REPORT

ZÜBEYİR BODUR

21702382

Spring 2020

1. Introduction

In this project, a checksum calculator with a memory of words were asked to be implemented. To implement such a calculator, HLSM approach is used for design and a register file was necessary for memory. Of course, other types of combinational logic were also used.

In FPGA, same buttons and switches are used as asked in the project description. SW15-8 were bits of 8-bit data and SW3-0 was the bits of the address for this data to be entered. Up push button is used to calculate checksum and display it in hexadecimal, right/left push buttons are used to switch the display between words in the memory, down push button is used to show number of words used in the checksum calculation and center push button is used to enter the data in SW15-0 to address SW3-0.

Almost all parts are successfully implemented in this project expect one. We were asked to have a reset state on this machine (part 6) but unfortunately I failed to implement such a state in my FPGA. Instead of words having data 0x00, 0x01, 0x02…, 0x0f, they all were initialized as 0x00 in the FPGA. However, in my simulation, they were successfully initialized as asked.

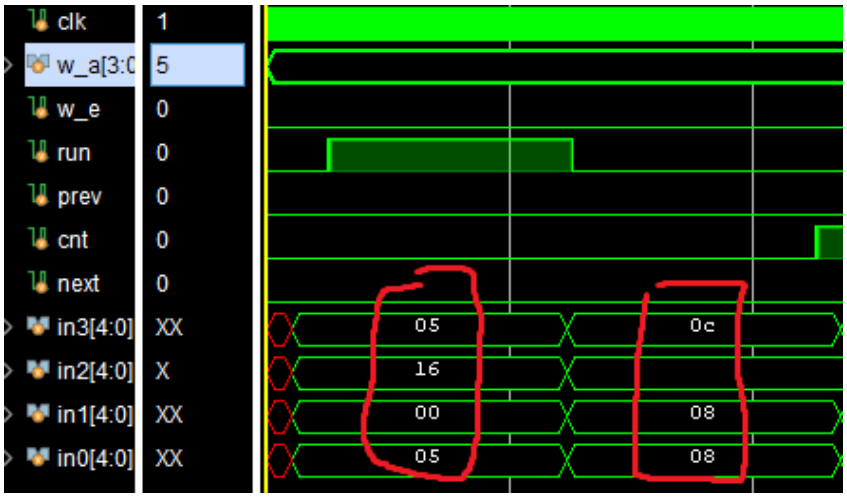


Figure 1: Simulation for the main module of the project. Outputs in3-0 are 5-bit signals representing a hexadecimal number or a symbol, and local variables of this module. After a couple of nanoseconds, display shows the address represented by the switches. In here, switches are initially set to 1001, so 5-05 is displayed initially. Also, checksum is calculated as 88, as calculated in the project description.

2. Block Diagrams

The solution for this project can be separated into 2 main parts. One of the main parts is calculating the checksum & count for the current state of the memory. The other is implementing an interface for displaying checksum & count and switching between words. Let's call the first one "Model HLSM" and the second one "View Logic HLSM". All diagrams & schematics are after the written descriptions, in page 5-11.

a. Model HLSM

This design is created to calculate the checksum & counting words for the current state of our memory array. If run button is pressed, system will start processing words, count the number of the words and calculate 2's complement for the sum of all words. At the end, it will also inform the View Logic HLSM that checksum calculation is done.

Inputs: run (bit), r_d_0[16] (8 bits)

Outputs: checksum , num_of_cycles (8 bits); over (bit)

Local Storage: sum, checksum_reg (8 bits); i (5 bits)

b. View Logic HLSM

This design is created to control the seven segment display according to user's decisions. It involves counting 10 seconds for displaying checksum & count, memory display mode, changing the data displayed using display next/previous buttons and any other things that change the numbers displayed in the seven segment display.

Initially, the system will start with the memory display mode. Memory display mode means displaying the data whose address is represented by the switches SW3-0. After processing the address to the memory and receiving the data, it is displayed. Then, there will be 6 options; user presses next/previous buttons, user presses run button, user presses display counter button, user presses enter data button or user doesn't do anything.

Since this one is complex enough, datapath and controller FSM is not included below.

Inputs: w_e, run, prev, cnt, next, over (bit);

r_d_1[16], checksum, num_of_cycles (8 bits)

Outputs: in3, in2, in1, in0 (5 bits)

Local Storage:in3_reg, in2_reg, in1_reg, in0_reg (5 bits); r_a_1(4 bits); ten_C (32 bits)

i. Display Loop

If user presses next/previous button read address of the first port will be incremented/decremented. If user presses enter data button, address won't be changed. Both in these 3 options, and in the init state too, system will enable reading and then display the current word. After displaying, if either of these buttons are pressed, the system will go through the same steps. This can be called as display loop.

ii. Checksum & Counter Loop

If the user presses run button, we wait for Model HLSM to calculate the *checksum*. If *over* is HIGH, this means *checksum* is ready and can be displayed. Then, we clear the counter, and in the next clock cycle, we display checksum and count up to 200,000,000 (clk freq is 20MHz). While counting still goes on, if any of the push buttons (except enter data) is pressed, we change the display accordingly. Also, if count reaches 200,000,000 , system goes to init state.

If the user presses display counter button, exact same process happens as the checksum except two things. One, there is no need to wait for Model HLSM as we don't care about whether checksum calculation is done; two, the digits to be displayed are different.

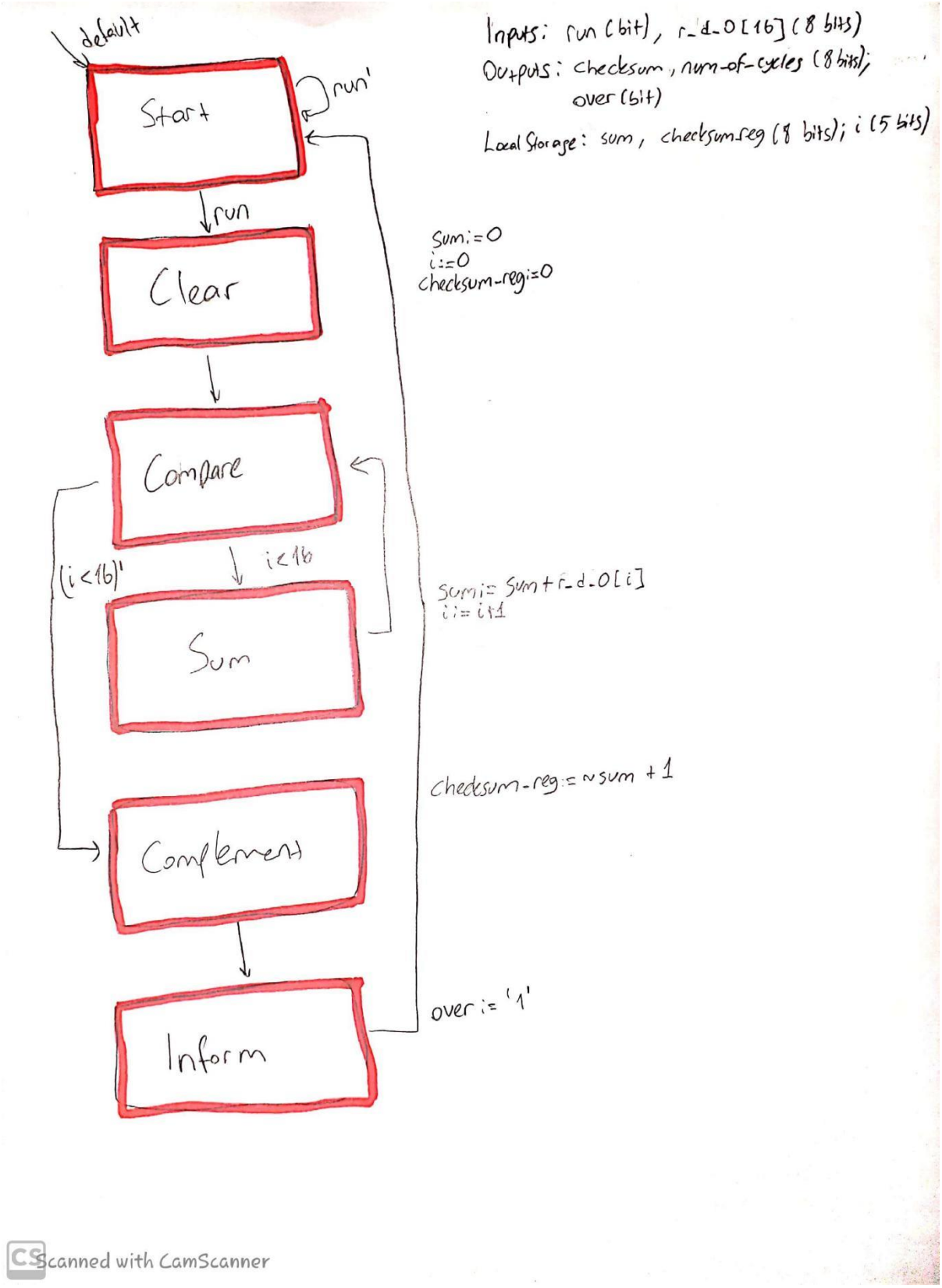


Figure 2: State Transition Diagram for Model HLSM

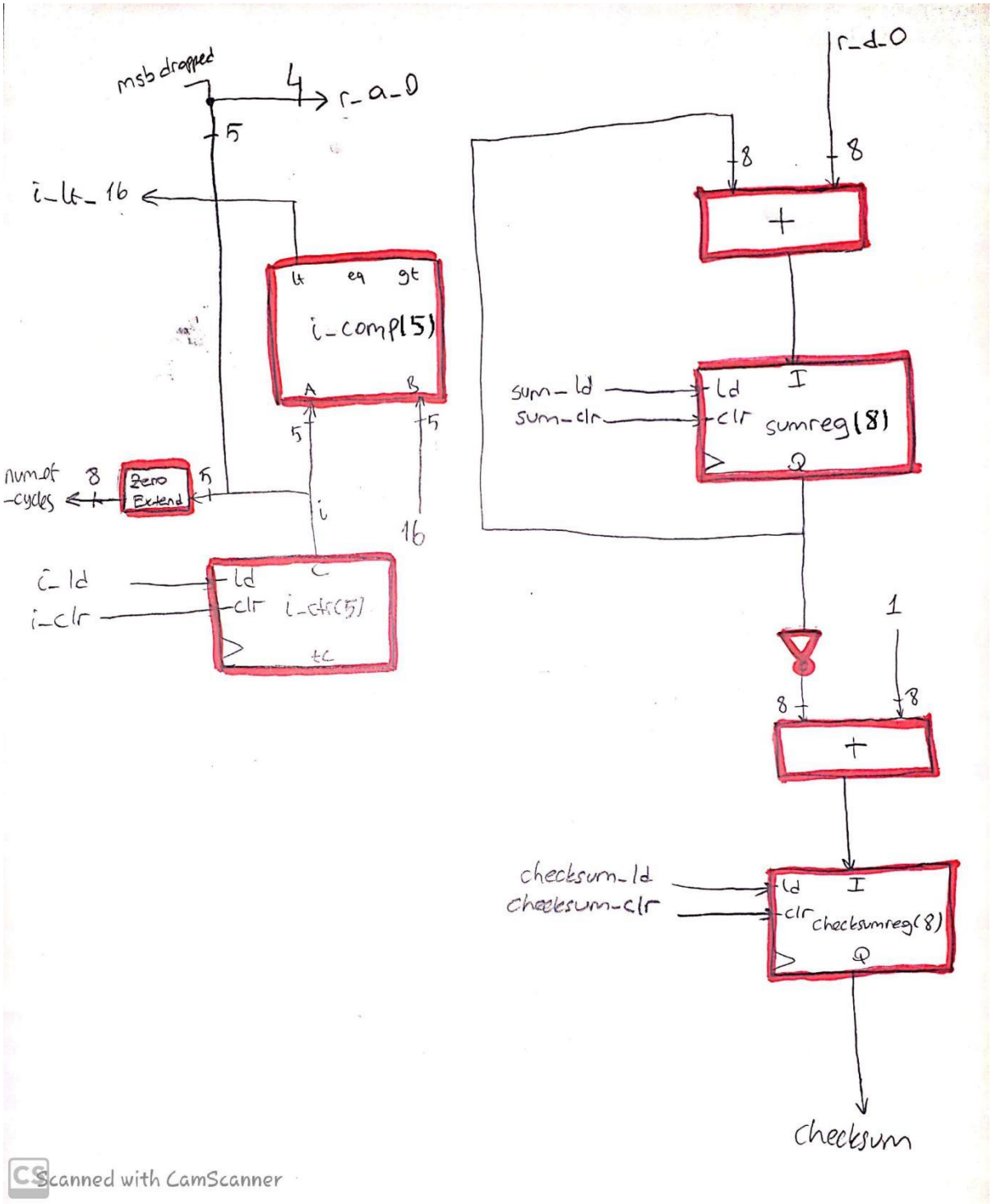


Figure 3: Datapath components for Model HLSM

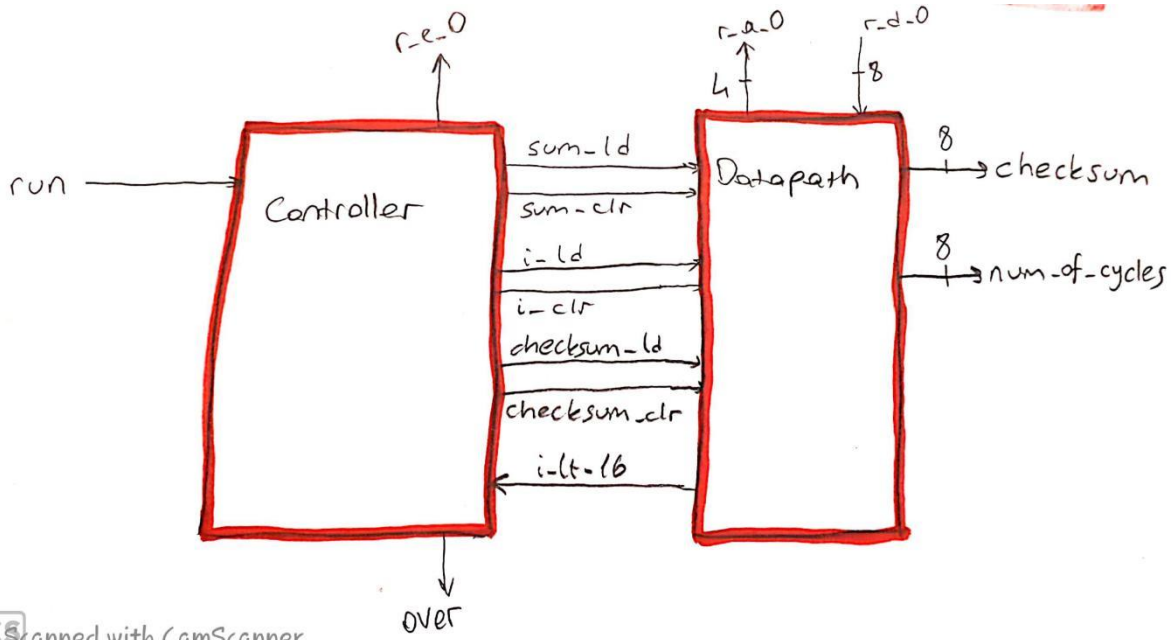


Figure 4: Connection of controller and datapath for Model HLSM

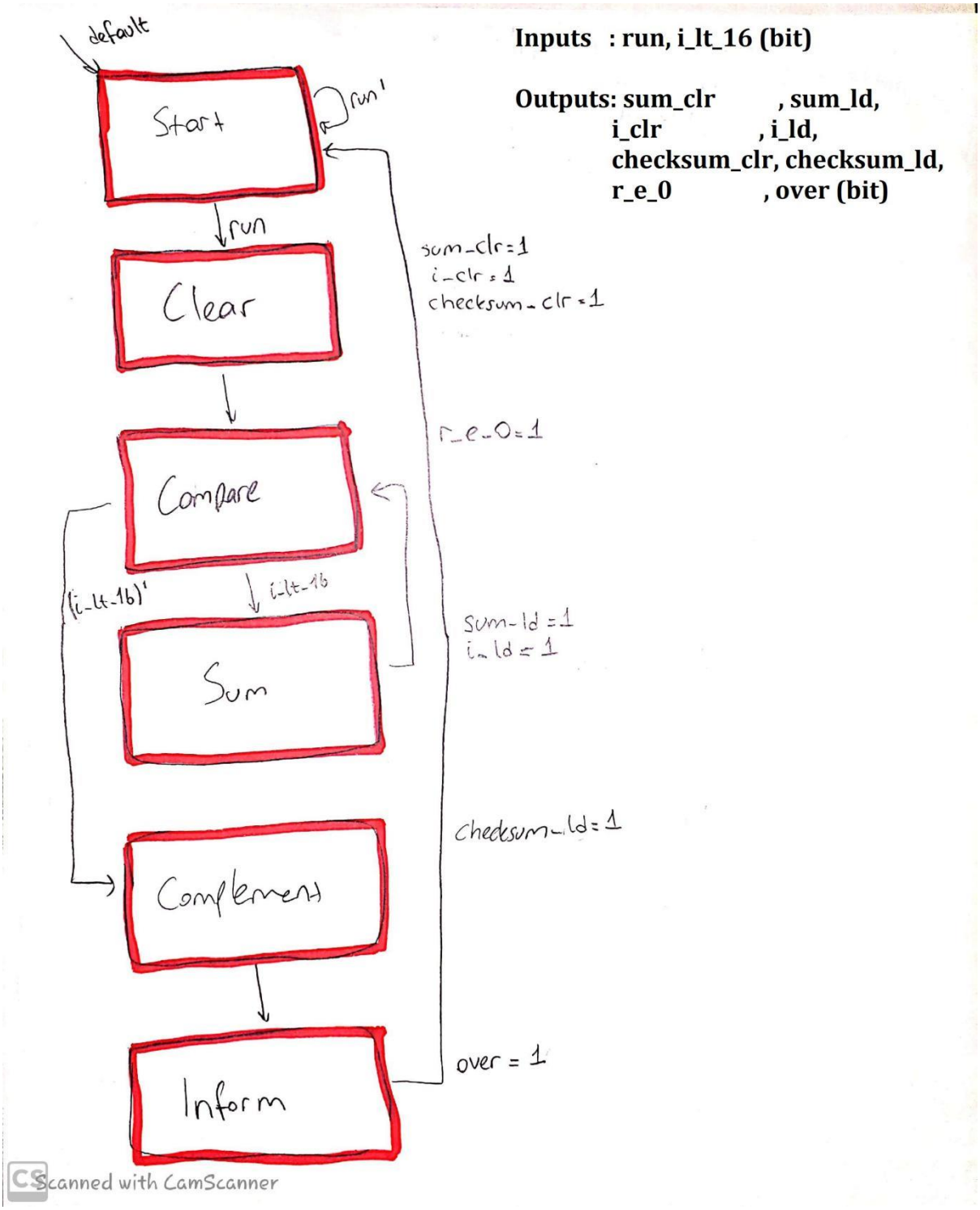
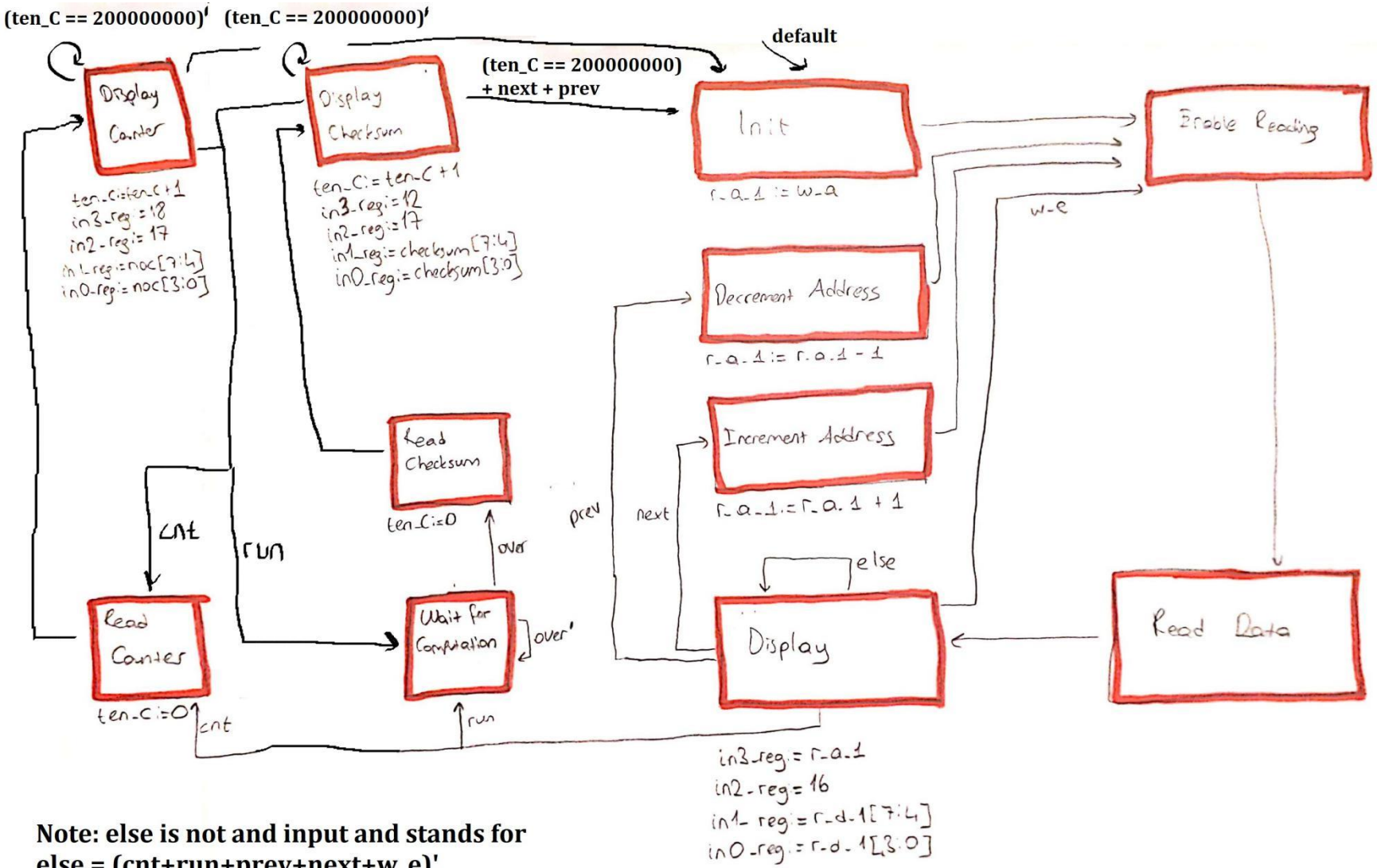


Figure 5: FSM for the controller of Model HLSM



Note: else is not and input and stands for
else = (cnt+run+prev+next+w_e)'

Figure 6: STD for View Logic HLSM

3. Detailed Explanation of Work

a. Important Modules

In this project, 2 important HLSMs mentioned above are used. Both are connected to each other via the *over* signal. Those 2 were written in 4 different modules and these 4 are combined in the module called *main*. *main* also contains the module *SevSeg_4digit*, which converts 4 4-bit numbers into real signals in FPGA.

i. Model HLSM

3 of these modules belongs to the Model HLSM, which stores the memory, calculates the checksum and number of words for the current state of the memory. The modules used in this one are called *regFile*, *controller* and *datapath*.

regFile module is a 16x8 register file, has 2 read ports and 1 write port. Those ports have 4-bit address, 8-bit data and an enable signal. These are called 4-bit *r_a_0, r_a_1;* 8-bit *r_d_0, r_d_1;* single bit *r_e_0, r_e_1;* 4-bit *w_a,* 8-bit *w_d* and single bit *w_e* respectively.

This register file module is a little different than a simple one; it has two states in order to reset the memory in the first clock edge. Name of the states are *init* and *read_write*. Default state is *init,* and in this state, the system initializes the memory array with the content asked in the lab description, and goes to the state *read_write* on the next clock edge. In *read_write* state, enable signals are checked for each port and if they are HIGH, reading/writing happens. As previously told in the introduction page, this approach for resetting the memory array works only in simulation.

controller module is the controller for the Model HLSM. It has the *run* button and *is_last_data* from the *datapath* as inputs; *i_clr, i_ld, sum_clr, sum_ld, checksum_clr, checksum_ld, ctr_clr, ctr_ld, r_e_0, over* signals as outputs. This module controls the sequence of computations happening in the *datapath*.

datapath module is the datapath for the Model HLSM. It takes the outputs of the controller, which are clear and load signals to a register/counter (*i_clr, i_ld*...), and the zeroth read data of the *regFile* (*r_d_0*) as inputs; 4-bit *r_a_0,* 8-bit *checksum,* 8-bit

num_of_cycles and single bit *is_last_data* as outputs. It takes orders from the *controller,* data from *regFile* and using these, computes the outputs.

These 3 modules are connected to each other and work like the following: *controller* controls the state Model HLSM is in, the clear/load signals of the *datapath* and *r_e_0* signal of the *regFile*. *datapath* sends appropriate address bits (as *r_a_0*, from 0000 to 1111) to *regFile*, takes the data from *regFile* (*r_d_0*) then computes the *checksum* for those words and computes the *num_of_cycles* as the computation continues. When the last data is reached, *datapath* informs the controller that *is_last_data* is HIGH. When computation is done, controller goes to the state *inform* and sets *over* signal to HIGH so that View Logic HLSM stops waiting for the computation and starts processing the *checksum* to display.

ii. View Logic HLSM

Other remaining module is implemented to control the state of the seven segment display, and is called *sevenSegmentLogic*. This module itself is an HLSM and is actually separated to two different parts by using comments, namely a controller and a datapath. This module has 11 states, a clock input, 10 inputs, 6 outputs and 14 local variables.

Being this complex though, what it does can be described as controlling the read port 1 of the *regFile* to switch between memory and outputting 5-bit *in3, in2, in1, in0* for the display. *in3-0* stand for a hexadecimal or a symbol. The dictionary for them is given in the next page. Since the state transition diagram of the View Logic HLSM is already drawn and explained in the section Block Diagram, we can move on.

Before moving on, let us note that this module produces 5-bit outputs for seven segment display to use, but it doesn't send them to the display directly. Instead, it sends them to the module called *SevSeg_4digit*.

iii. Converting Hexadecimals to Real Signals

SevSeg4_digit takes these 4 5-bit signals and converts them to active low signals to display in FPGA. It actually shows only 1 digit in one clock cycle, and doesn't display other digits. This is because FPGA is not able to display different digits in one clock cycle. For this problem, clock is divided to 1526Hz so that human eye would see it as if all of

them were shown at the same time.

This module is also a simple HLSM making use of “5 to 7” decoders. It has in3-0 as inputs, and a, b, c, d, e, f, g, dp, an (actual signals in FPGA) as outputs. To explain simply, this module helps rotate 4-bit an signal in one hot mode. It is 1110 initially so that rightmost digit is displayed first. Then, in each clock cycle, it is rotated one bit left. In addition, the digit to be decoded is also changed when rotation happens. Below is the look up table about what 5-bit signals correspond in the display.

inX	digit/symbol	inX	digit/symbol
5'd0 :	0	5'd10 :	A
5'd1 :	1	5'd11 :	B
5'd2 :	2	5'd12 :	C
5'd3 :	3	5'd13 :	D
5'd4 :	4	5'd14 :	E
5'd5 :	5	5'd15 :	F
5'd6 :	6	5'd16 :	-
5'd7 :	7	5'd17 :	=
5'd8 :	8	5'd18 :	c
5'd9 :	9		

iv. Combining the Work in a Single Module

In the module *main*, all of these 5 modules mentioned above are combined, along with a *clockDivider* and five *buttonDebouncer* modules for each push button. In this module, the clock signal that is necessary for all sequential logic is divided by 5, except *SevSeg4_digit* because it has its own clock divider. This clock division reduces the clock frequency of the remaining modules to 20MHz. In addition, inputs coming from push buttons are smoothed using *buttonDebouncer*. Finally, this module connects switches used in FPGA to their LED’s so that it is easier to see which switches are on.

v. Button Debouncer

buttonDebouncer module doesn’t totally fix the button debouncing problem seen in FPGA’s. What it does is converting a lengthy HIGH signal into a single HIGH pulse. In other words, it helps push button input to fit in a single clock cycle.

It takes a clock signal, an input *pb_in*, and has an output *pb_out*. It has two D-Flip Flops and one AND gate. *pb_in* comes to the first flip flop and Q signal of this flop is sent

to D signal of the second flip flop. Then, Q' of the second flip flop is ANDed with Q of the first flip flop and the output of the AND gate is connected to *pb_out*.

b. Other Modules

Remaining modules in this project are all parametric modules and are digital building blocks. These are multiplexers, a register, counters and a clock divider. The parameter N in these modules controls the bit width of the important input & output.

clockDivider has N = 32 as default. The difference between this module and a standard clock divider (that uses a down counter with parallel load) is how the down counter is cleared. It has two states, *init* (default state) and *divide,* and the *downCounter* module. In *init* state, *downCounter* is cleared and nextate is *divide*. In *divide* state, *clr* of the counter is set to LOW and nextstate is *divide* at all times. This two states are used so that *clockDivider* is cleared in the very first clock cycle.

downCounter has N = 32 as default. It is an N-bit down counter with a parallel load. *ld* has priority over *cnt* in this module and it uses an N-bit *register*.

upCounter has N = 4 as default. It is an N-bit up counter with synchronous load and clear. It uses an N-bit *register*.

register has N = 4 as default and is an N-bit register with synchronous load and clear.

mux2 has N = 8 as default and it is an N-bit 2:1 multiplexer. *mux4* has also N = 8 and it is an N-bit 4:1 multiplexer. *mux4* is created by using 3 *mux2* modules.

4. References

buttonDebouncer module in main.sv. The sources in the following page were read to implement such module: https://www.fpga4student.com/2017/04/simple-debouncing-verilog-code-for.html . Accessed 8 May 2020.

SevSeg_4digit.sv, 2019-2020 Fall Semester, Lab material supplied by the instructor.