Design of a Sorter

By
Simon Dib
&
Nicolas Ghandour

Presented to
Dr Rafic Ayoubi

A Project for the Logic Circuits Course CPEN212

Faculty of Engineering
University of Balamand

10/12/2023

1 – Objective:

The main objective of the project is to implement the odd-even algorithm on a 16x8 register file filled with random (unsigned) numbers through designing the necessary logic circuits.

2 – Introduction:

Digital logic design is mainly about using transistor-based components called “logic gates” having voltages inputs and outputs that can be represent as binary numbers ‘0’ and ‘1’. In telecommunication, digital design showed a great efficiency and accuracy in transmitting and receiving signals by converting analog signals to digital signals (a sequence of zeros and ones), which later on made digital communication replace analog communication. Most importantly, digital design’s power stems in its ability to design larges electrical systems component, and hardware (physical part of computers) based on pure logic thinking and boolean (which deals with binary numbers) algebra.

3 - Description of the project:

The design aims to do three operations on the 16x8 register file: 1) to fill it with random numbers, 2) to sort the numbers in the increasing order using odd-even sorting algorithm, and 3) to read the numbers from all registers of the register file. Each of these operations was used as a black box and was enabled by the mother FSM based on the user’s input. Below is a table showing the corresponding select line input s[1:0] for each function:

s[1]	s[0]	Operation
0	x	Read all numbers
1	0	Fill with random numbers
1	1	Sort

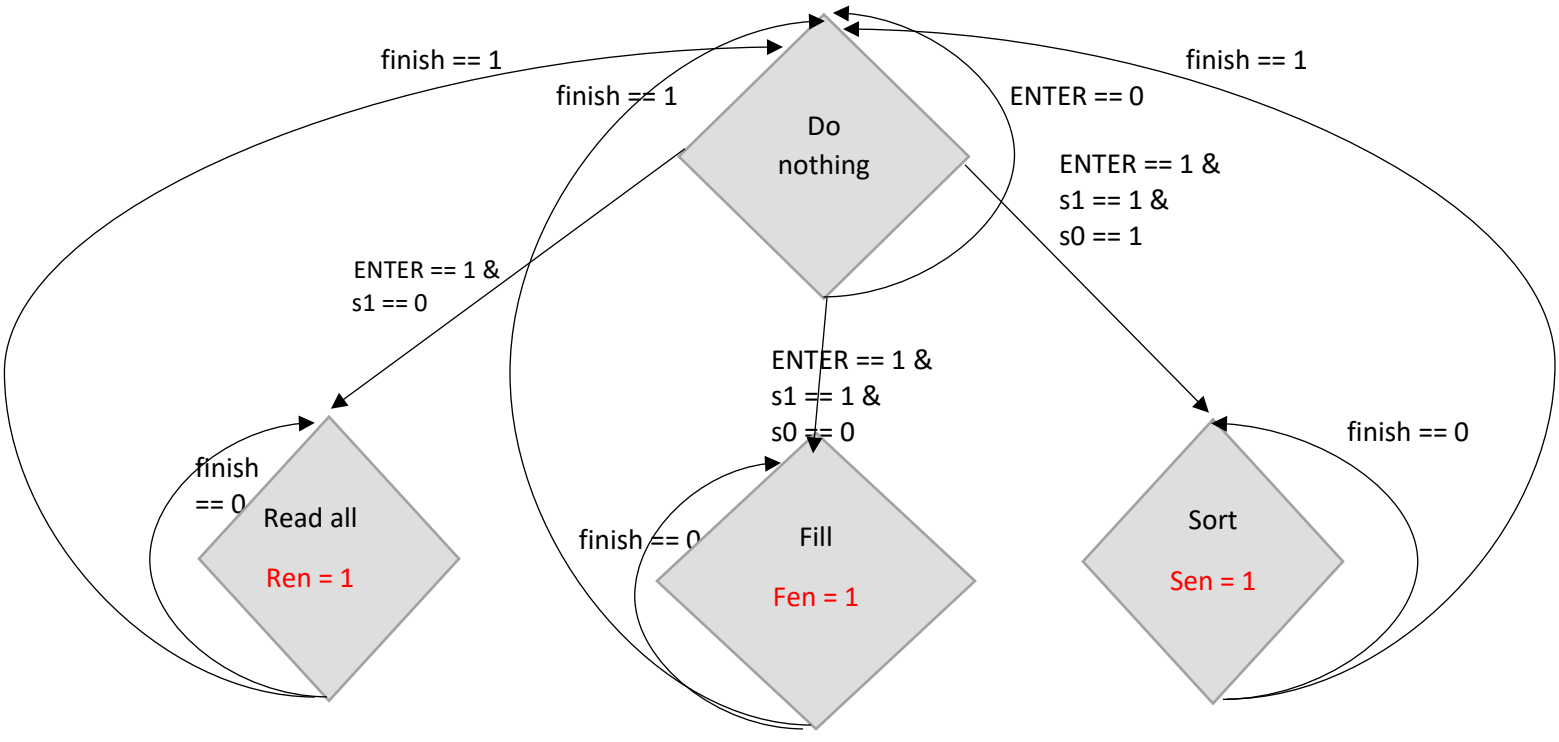
A - Mother FSM:

Inputs	Description
ENTER	key press
S[1:0]	Select lines for the operations
finish	Pulse indicating that the running operations has finished

Outputs	Description
Ren	Reading operation enable
Fen	Filling with random numbers operation enable
Sen	Sorting enable

The FSM works as follows:

To do an operation the user has to choose to the appropriate value of s[1:0] and press ENTER (ENTER = 1). The FSM will only consider the user's choice of operation once 'finish' = 1 is seen; otherwise, the choice is neglected by FSM.

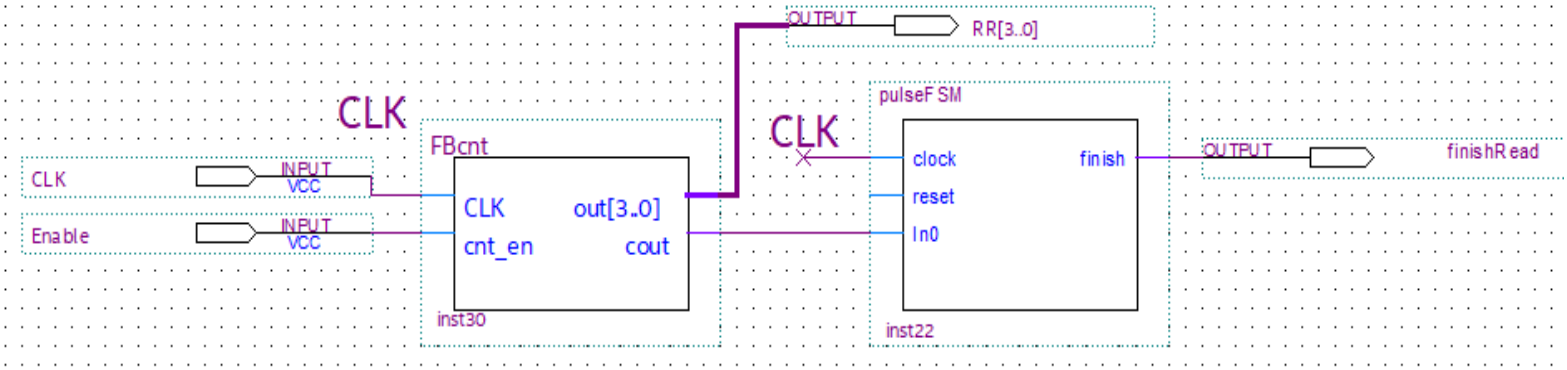


B - Operations:

I – Read:

Inputs	Description
CLK	clock
Enable	enable

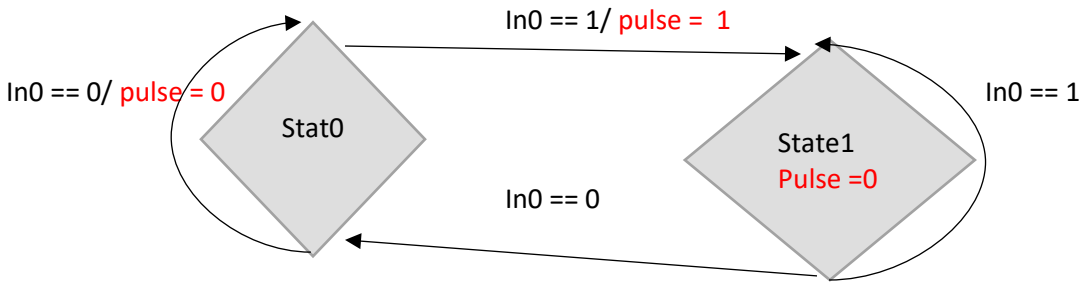
Outputs	Description
RR[3..0]	Counter output that will be connected later on to RR1[3..0] of the register file
finishRead	It signals the termination of the operation



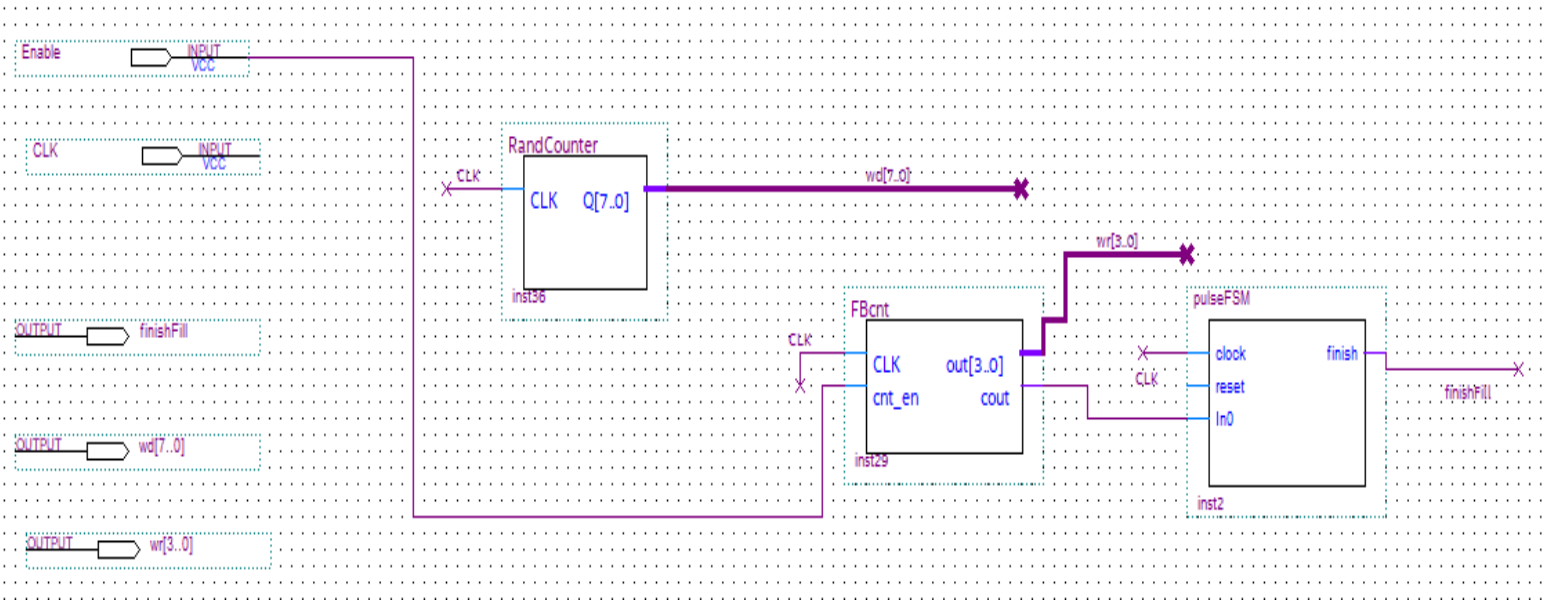
(For all the FSMs in the report, the outputs hold '0' if they're not mentioned in the states or at the transitions.)

Pulse FSM:

Function: it converts any long '1' into a pulse.



II – Fill with random numbers operation:



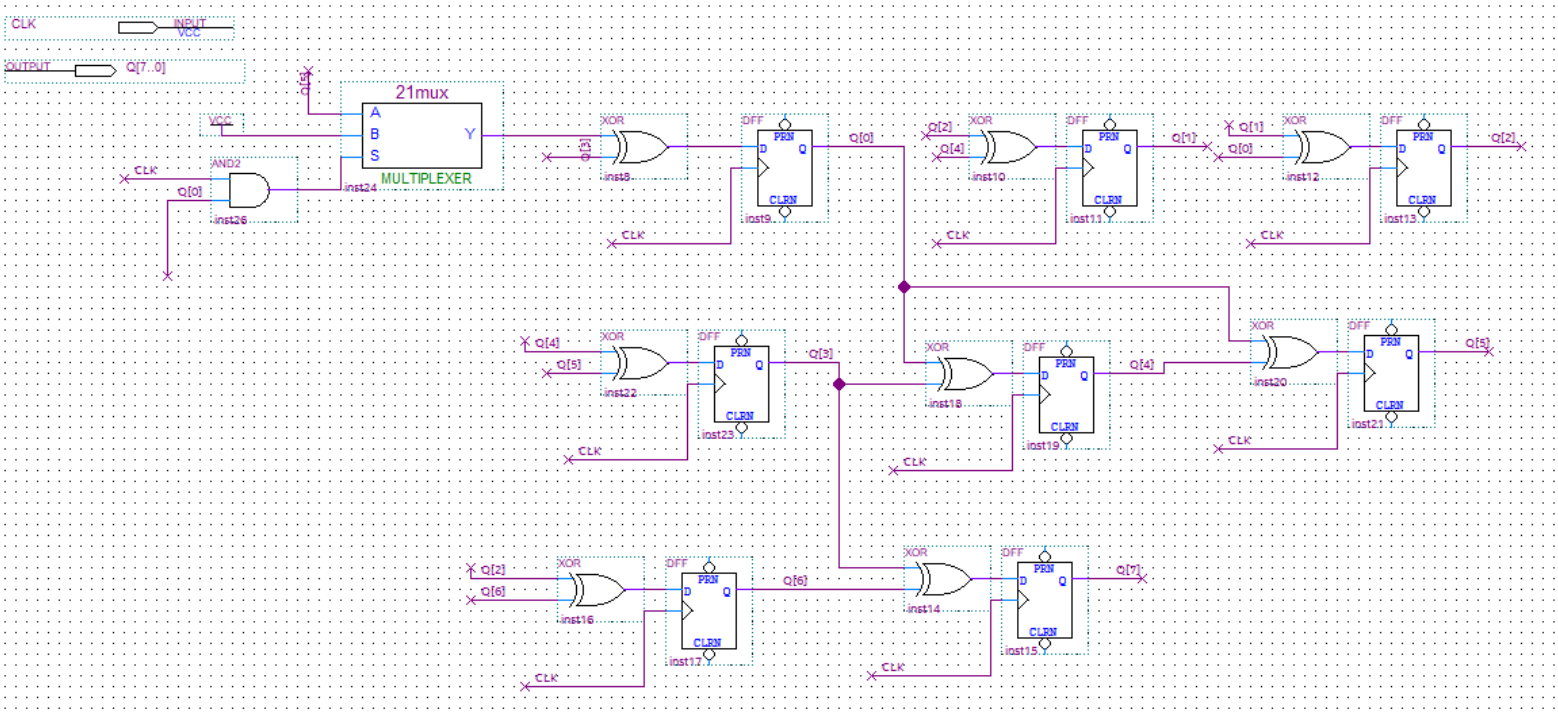
FBcnt ➔ 4-bit counter

RandCounter ➔ 8- bit Random counter

Inputs	Description
CLK	clock
Enable	enable

Outputs	Description
wr[3..0]	4-bit counter output that will be connected later on to the WR of the register file
Wd[7..0]	8-bit random counter output that will be connected later on to the WD of the register file
finishFill	It signals the termination of the operation

RandCouter



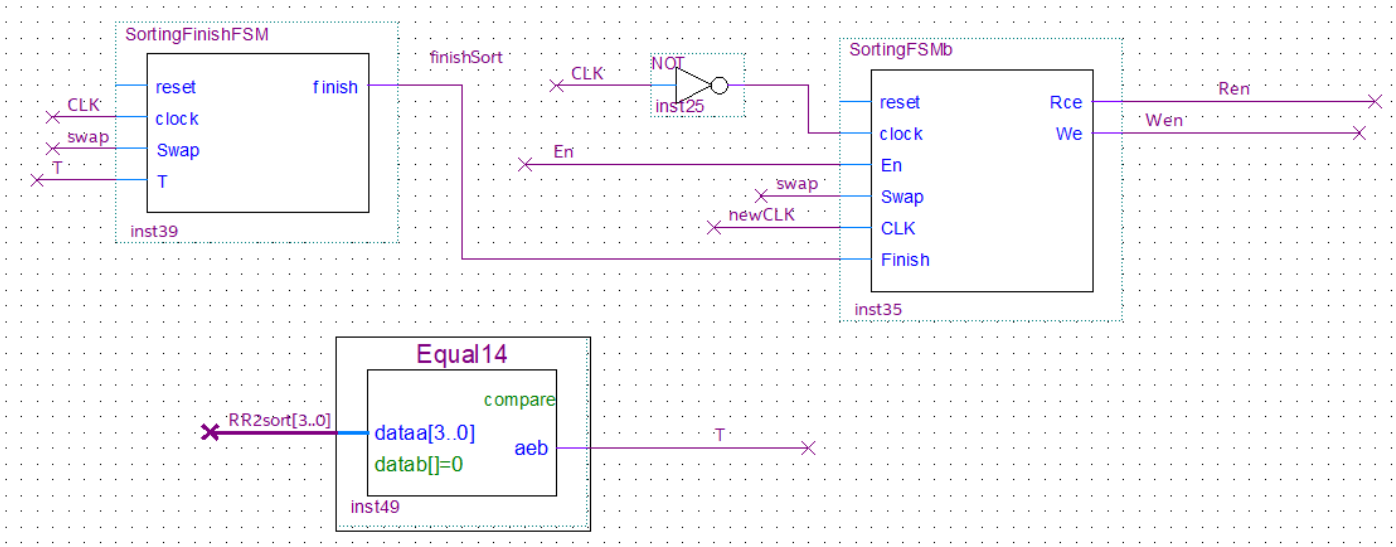
III – Sort operation:

Inputs	Description
CLK	clock
En	Enable coming from the MotherFSM
RD1[7..0]	Read-data port 1
RD2[7..0]	Read-data port 2

Outputs	Description
Wen	Write enable that will be connected to WE of the register file
WDsort[7..0]	8-bit data output that will be connected to the WD of the register file
WRsort[3..0]	4-bit data output that will be connected to the WR of the register file
RR1sort[3..0]	4-bit data output that will be connected to RR1 of the register file
RR2sort[3..0]	4-bit data output that will be connected to RR2 of the register file

finishSort	It signals the termination of the operation

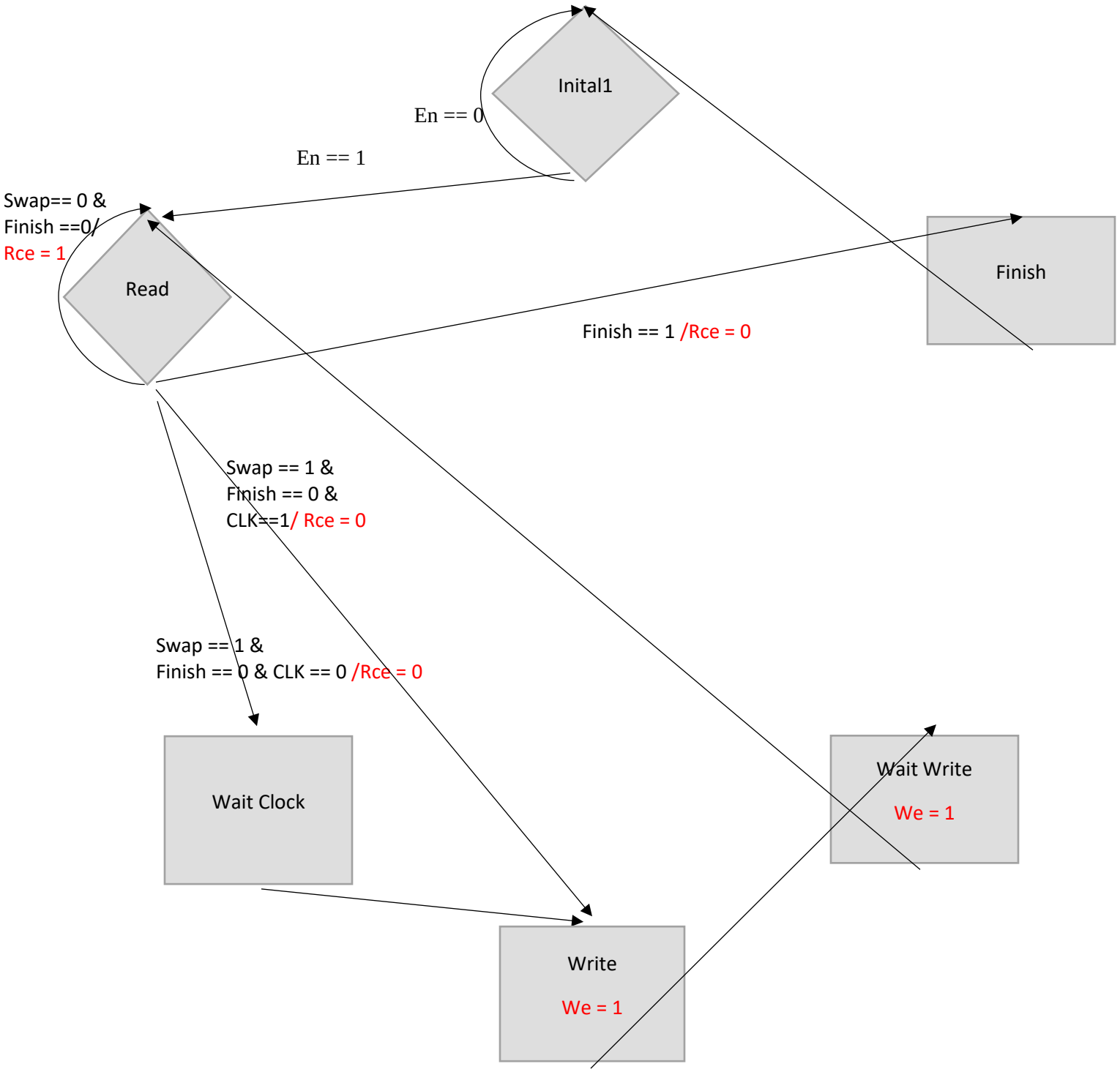
The sorting process works as follows: as long as data read from RD1 is less than that read from RD2, it keeps reading the registers and compare them. Otherwise, it will stop reading and swap the corresponding data. Once the counters tackle ‘T’ (two complete phases (even and odd)) with no swaps, the “finishSort” output will generate a pulse ‘1’.



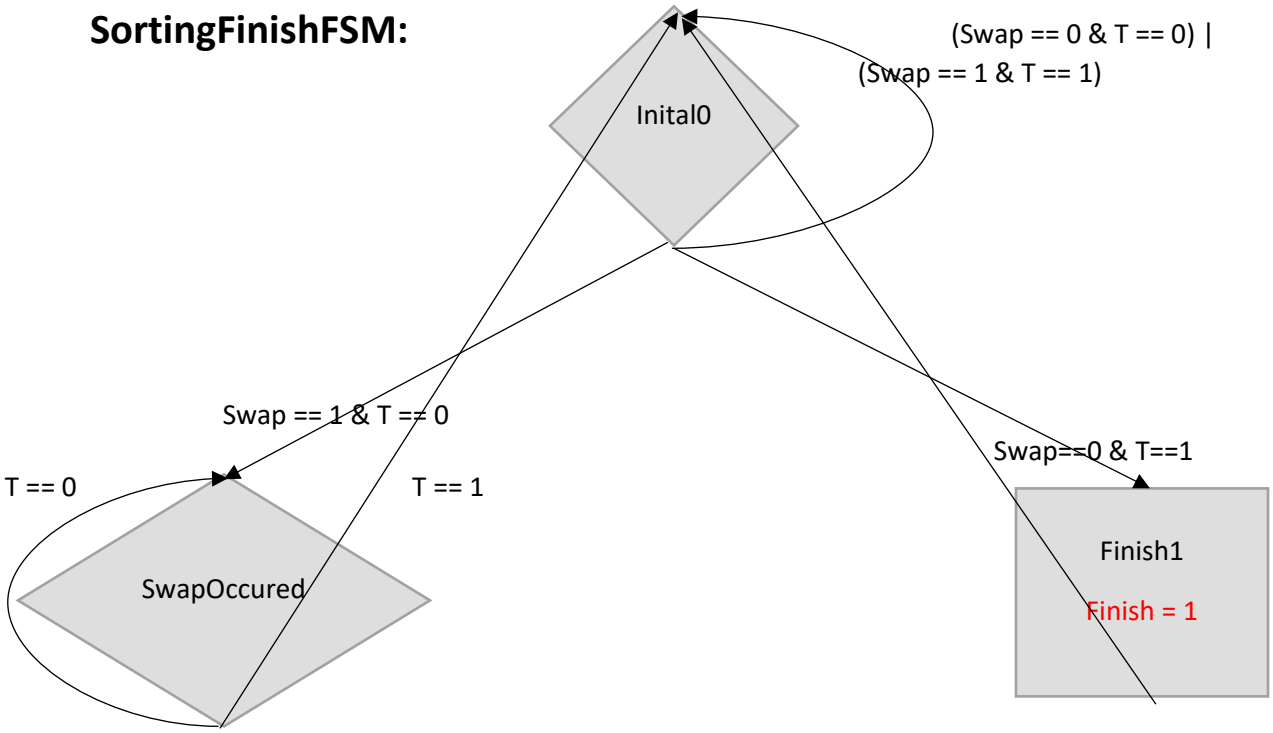
The inverter at the clock is put to ensure that the outputs be generated between falling edges so that other components in the circuit can receive these outputs as their inputs perfectly at their rising edge.

SortingFSMb (main FSM of the sorting operation):

Outputs	Description
Wen	Write enable that will be connected to WE of the register file that will control the writing part
Ren	Connected to the enable of the 4-bit two by two counter that will control the reading part

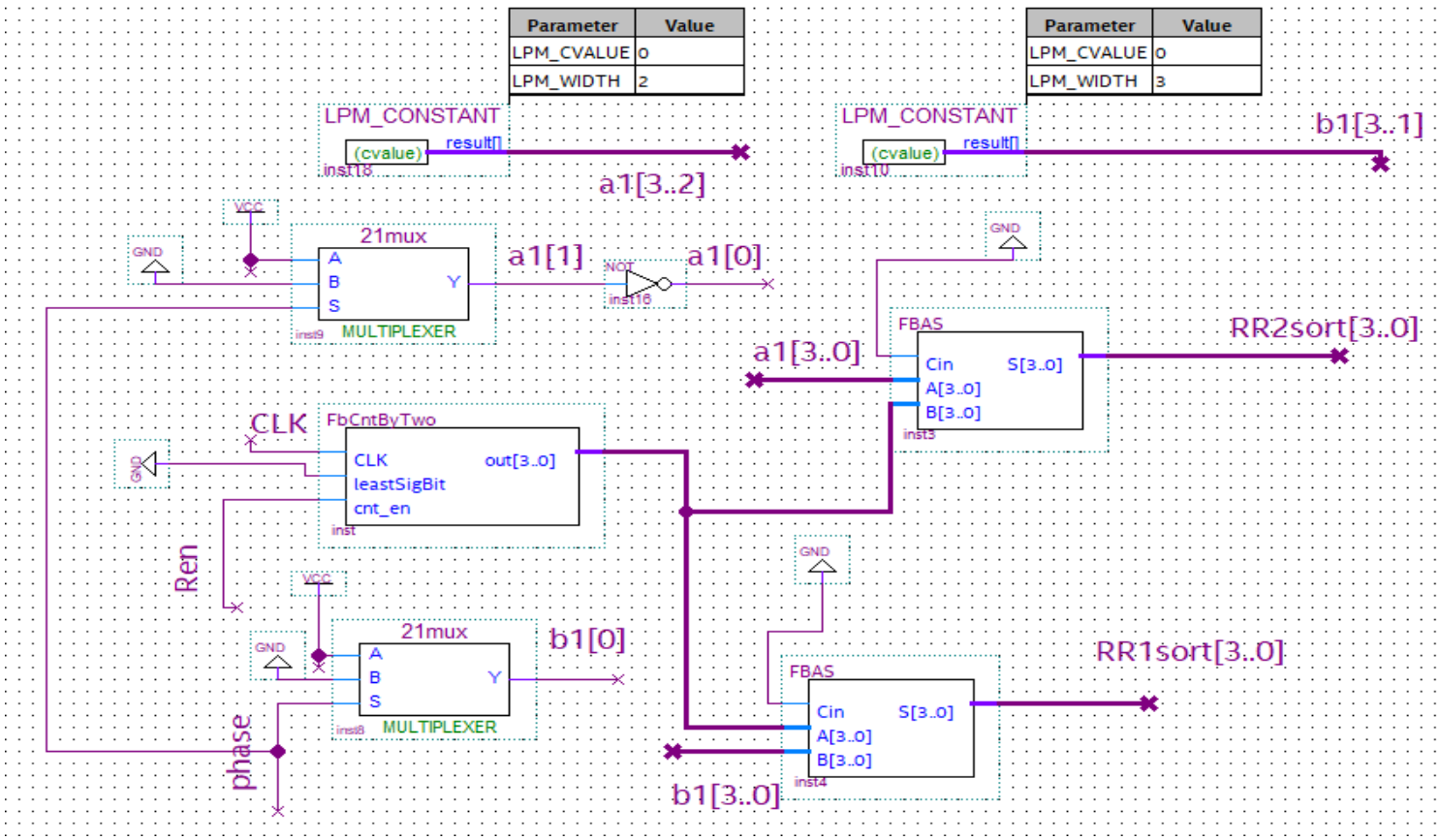

Inital1
En == 0
En == 1
Swap== 0 &
Finish ==0/
Rce = 1
Read
Finish
Finish == 1 /Rce = 0
Swap == 1 &
Finish == 0 &
CLK==1/ Rce = 0
Swap == 1 &
Finish == 0 & CLK == 0 /Rce = 0
Wait Clock
Wait Write
We = 1
Write
We = 1

SortingFinishFSM:



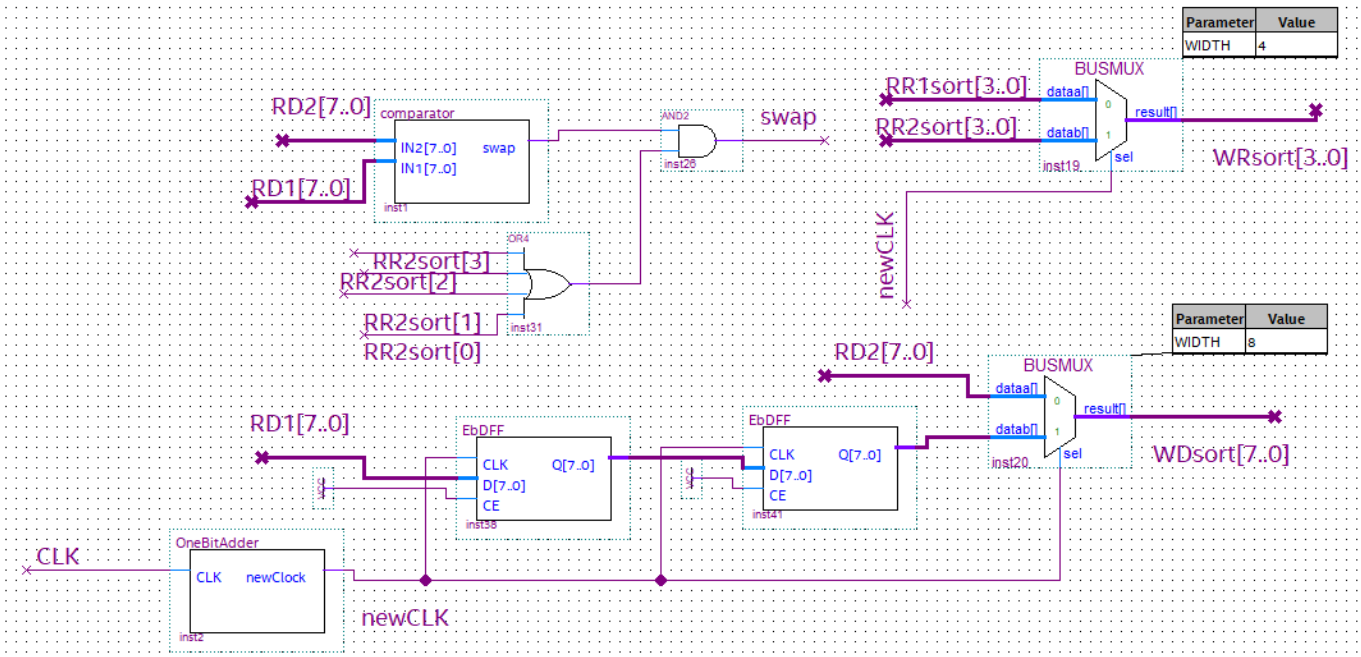
Reading part:

Two bus wires derived from the output port of a 4-bit two-by-two counter (FbCntByTwo) were inserted each at the input ports of two distinct 4-bit Adder/Subtracters (FBAS) (used as an adder in this case). At the second ports of the FBASs were connected two 2-to-1 MUXs such that the latter would generate (1 - 2) the output of the former (0 - 1) incremented by one, so that the two bus wires can act as two 4-bit two-by-two counters where one counts the even numbers and the other is counting the odd ones. The 'phase' select line of the MUXs plays the role of adjusting the starting value of the counter according to the phase (even or odd). The two bus wires were connected to RR1sort[3..0] and RR2[3..0].



Comparison and swapping parts:

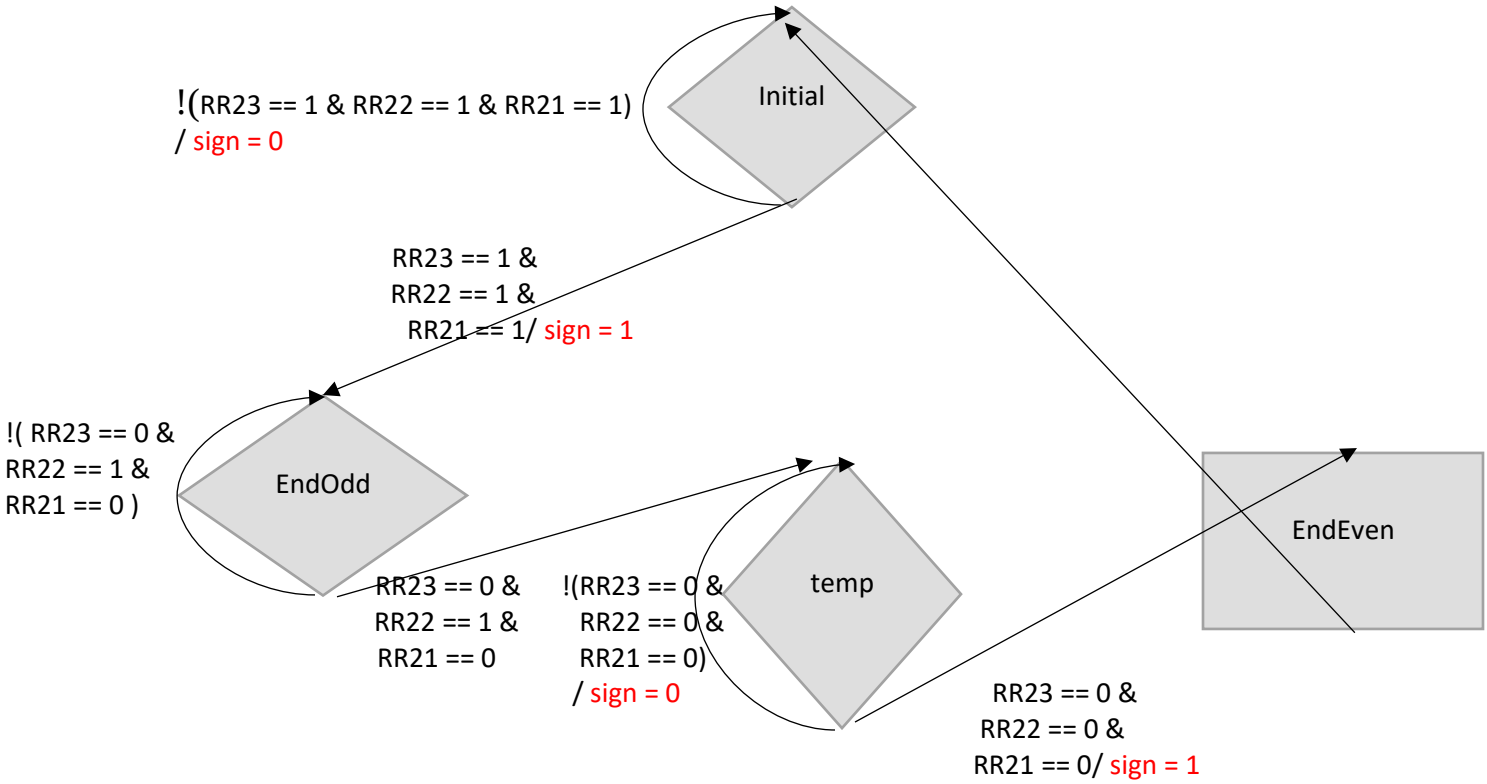
First, the data read through RD1[7...0] and RD2[7..0] will be inserted at the input ports of the “comparator” IN1[7..0] and IN2[7..0] respectively. The “comparator” internally is an unsigned subtracter (IN2 - IN1) and generates a “swap” signal whenever there is overflow (IN1 > IN2). During the transition from the even phase to the odd phase, the reading counters will pass through ‘15 – 0’ before resetting their values (so to count 0 -1, 2 -3, 4 -5 in the odd phase, and so on). Therefore, during the transition (when RR2 reaches ‘0000’) the counter should not stop to swap. Then, whenever a swap occurs, the reading counters will stop and the WE will be set to ‘1’. Two BUS- MUXs were used in this process when one holds the addresses of the registers (RR1 RR2), and the other holds their data inversed (RD2 – RD1). The two 8-bit DFF were added in order not to lose the RD1 when RD2 is being written in address RR1 (register where RD1 is initially saved). And having only one WD port in the register file, a “newCLK” (generated using a one-bit adder) was utilized to be able to write the data sequentially.



Phase-switching part:

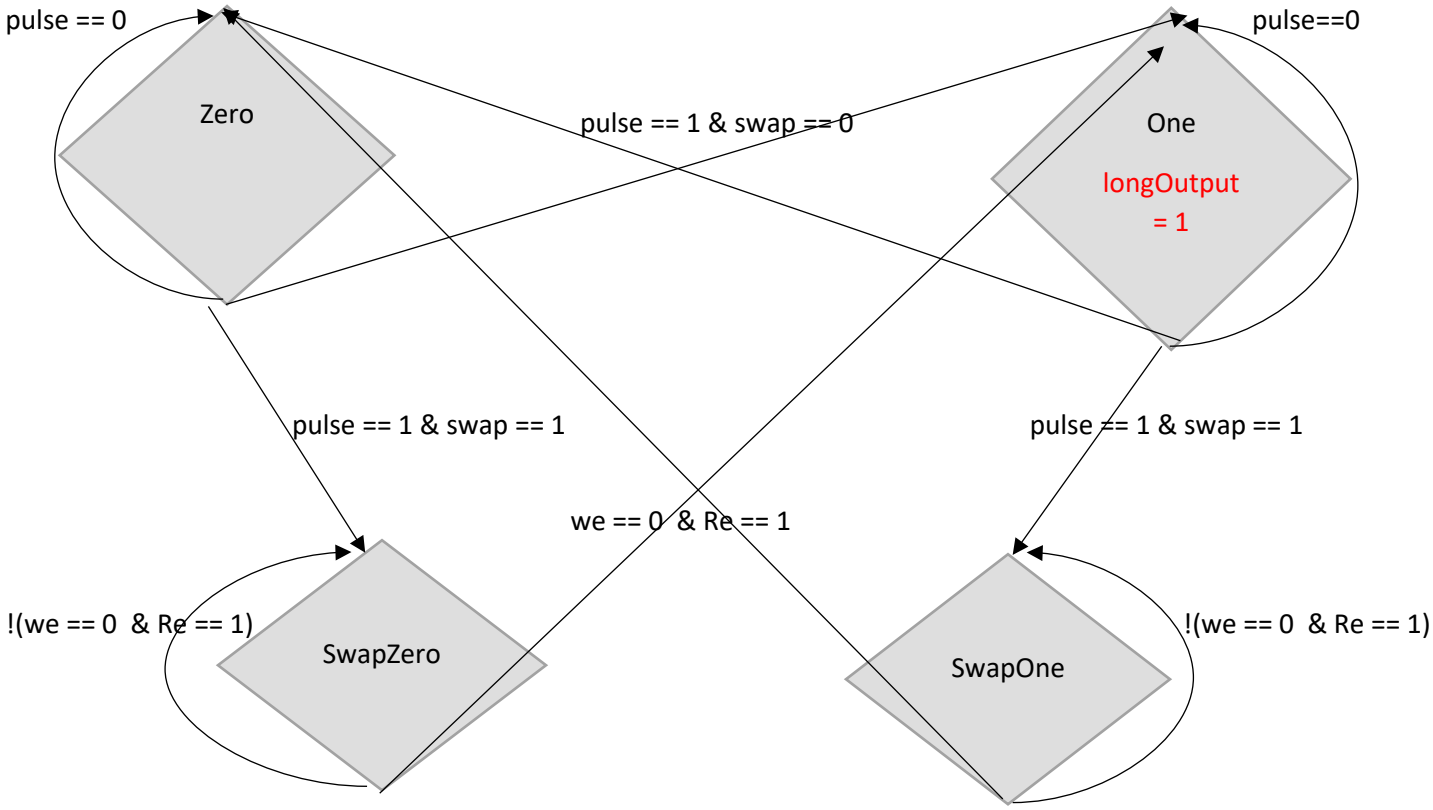
PhaseSwitchingbFSM

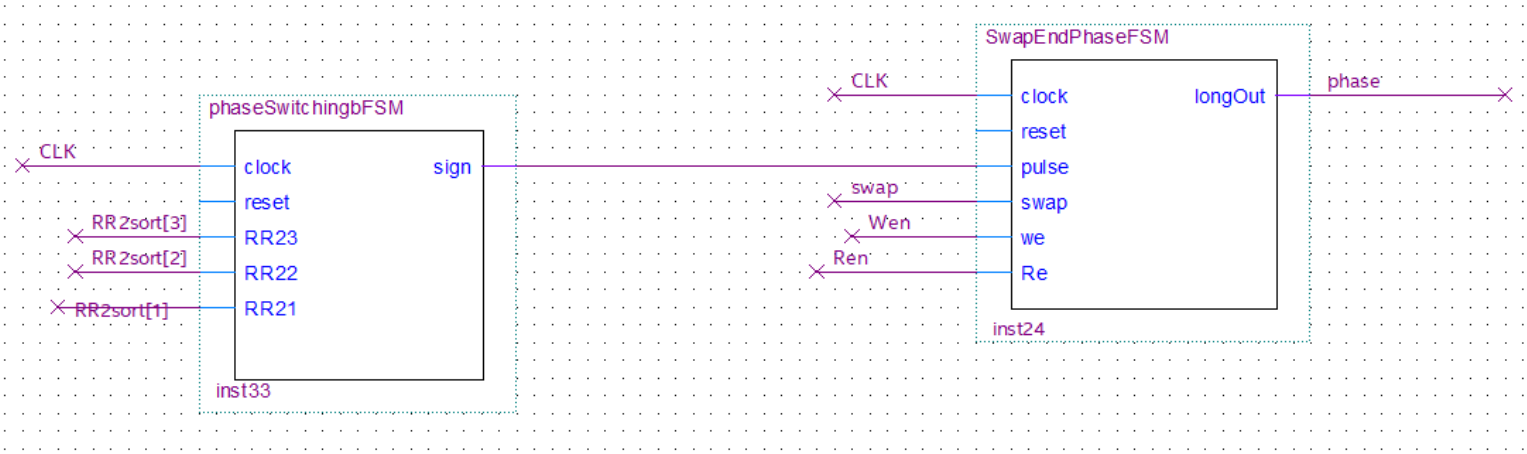
The function of this FSM is to generate a pulse once the reading counter reach the end of each phase (14-15 end of odd phase and 15 – 0 end of even phase). Since in this FSM we are neglecting the least significant bit, a strange scenario will occur: once RR2 reaches 15 (RR2[3:1] = ‘111’), the output pulse will be generated and the phase will change to even. But instead of waiting the whole phase before regenerating the output pulse, RR2[3:1] will directly pass through ‘000’ and generate a pulse, which is unwanted. That is why a new state “temp” was added to ensure that RR2[3:1] pass through ‘010’ (a checkpoint value indicating that RR2[3:1] passed the first ‘000’) first before rechecking when it reaches the end of the phase.



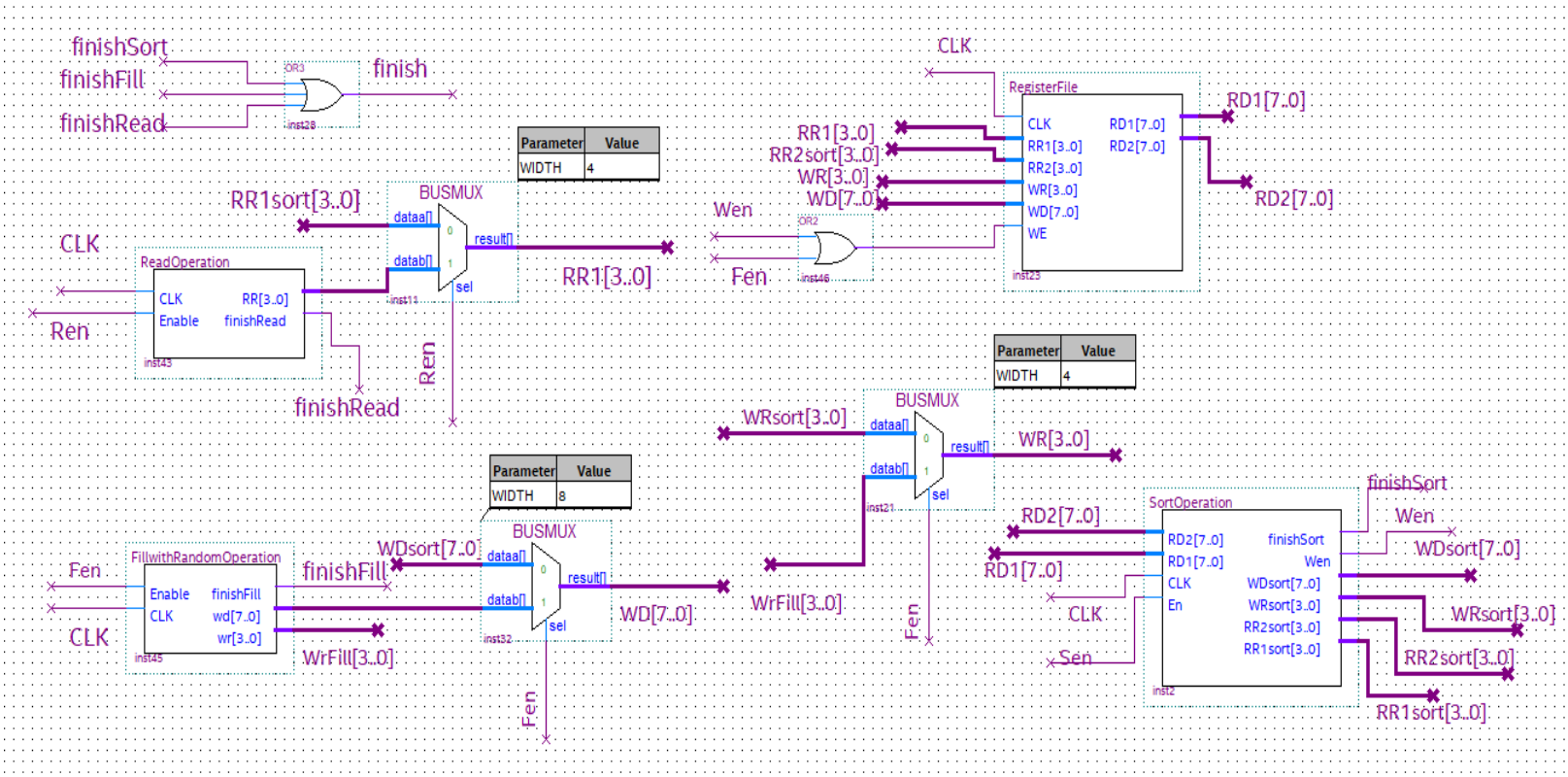
SwapEndPhaseFSM

The simulation revealed that when the reading counters reach “14 - 15” (end of the odd phase), the phase switching is occurring without checking whether in this combination there is a need to swap. Thus, the FSM’s function is to stop the counters at this combination (14-15) in order to be able to check and swap if needed, and then proceeding to the even phase. The simulation also revealed that “we == 0 & Re == 1” is the condition where the sorter have finishes the swapping of two numbers/.


pulse == 0
Zero
pulse == 1 & swap == 0
pulse==0
One
longOutput
= 1
pulse == 1 & swap == 1
pulse == 1 & swap == 1
we == 0 & Re == 1
!(we == 0 & Re == 1)
SwapZero
SwapOne
!(we == 0 & Re == 1)

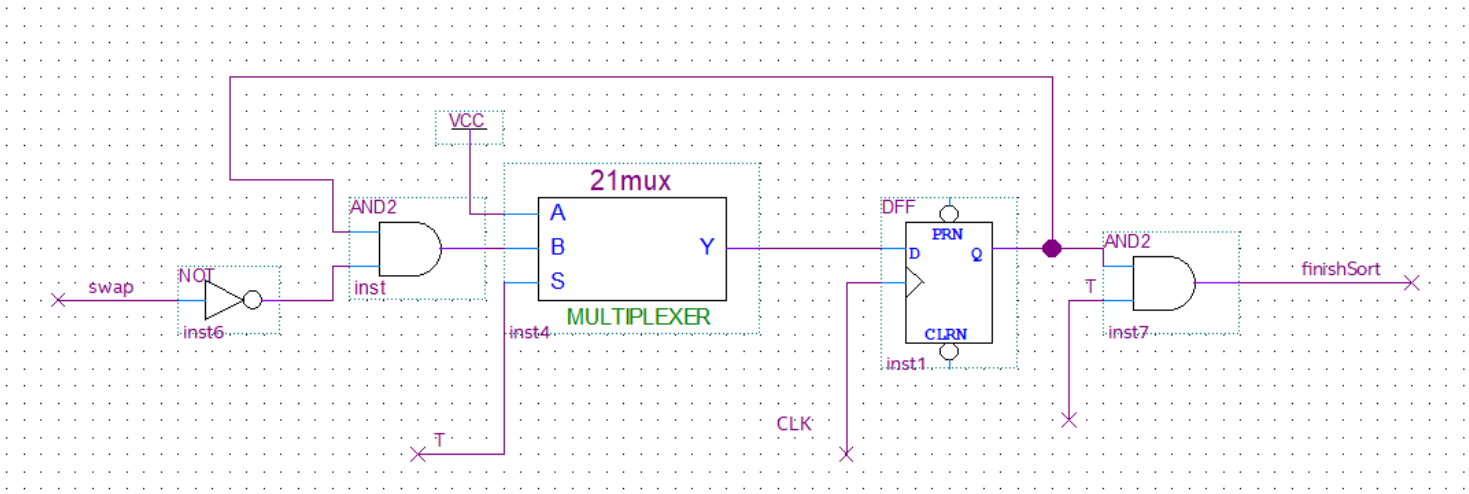

phaseSwitchingbFSM
CLK
clock
reset
RR23
RR22
RR21
sign
RR2sort[3]
RR2sort[2]
RR2sort[1]
inst33
SwapEndPhaseFSM
CLK
clock
reset
pulse
swap
we
Re
longOut
phase
swap
Wen
Ren
inst24

C –Main schematic:



6 – Alternative solution:

One alternative solution was to replace the SortingfinishFSM by the following design.

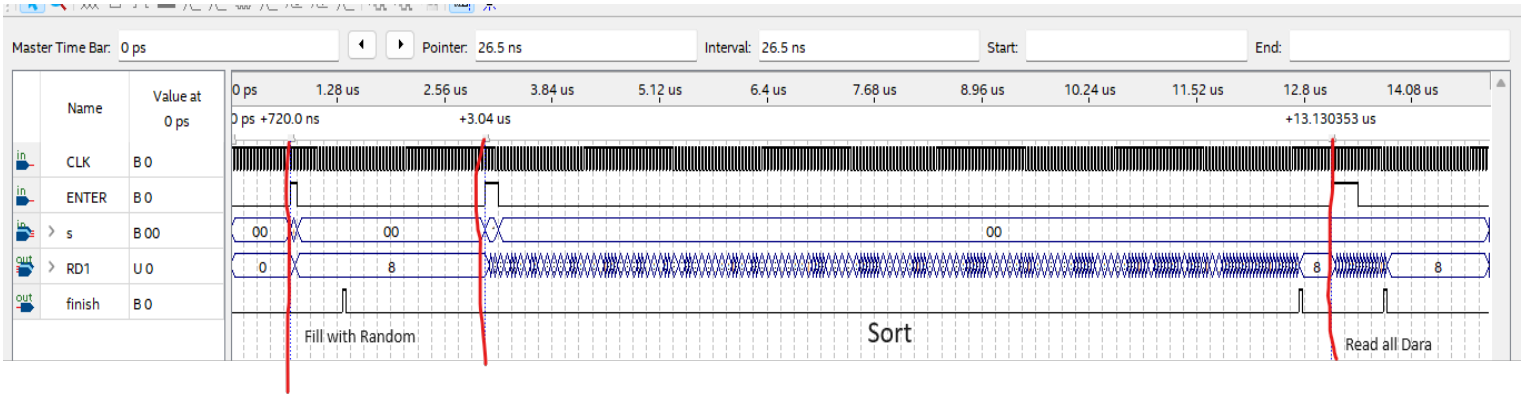


At the beginning of each T (period encompassing an odd and an even phase) the DFF is initialized to ‘1’ and when any swap occurs, the DFF will hold a ‘0’ until reinitialized. The DFF “finishSort” output will generate a ‘1’ if no swaps occurred along a whole T period.

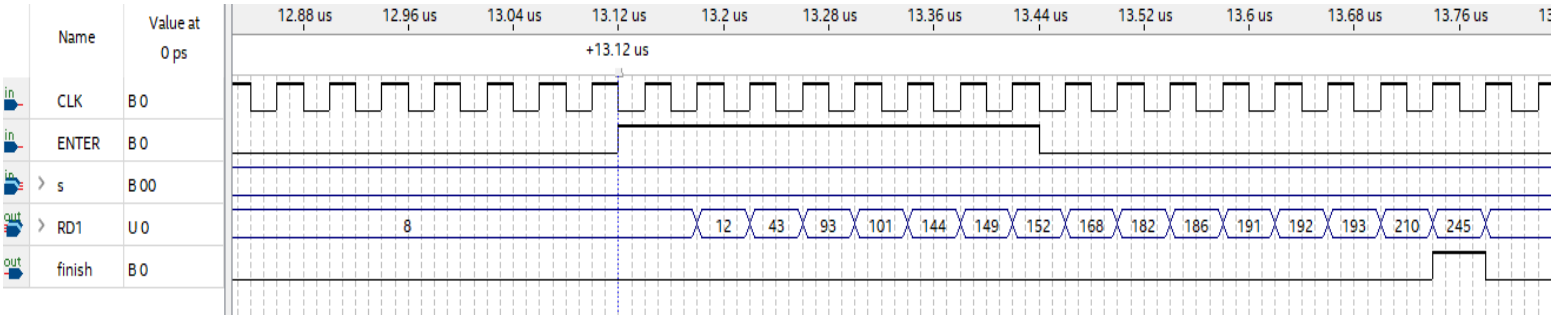
Being not much more efficient than the FSM, this solution was not implemented and the fact that FSM solution is easier to understand made us choose the FSM solution of the finish Sorting Part.

7- Simulation Waveforms:

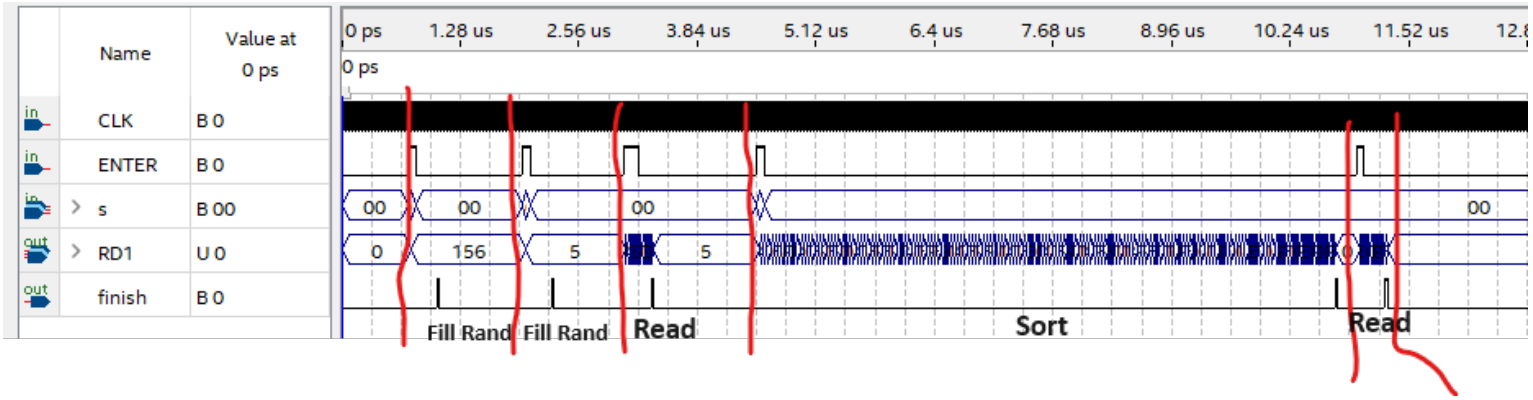
Waveform 1:



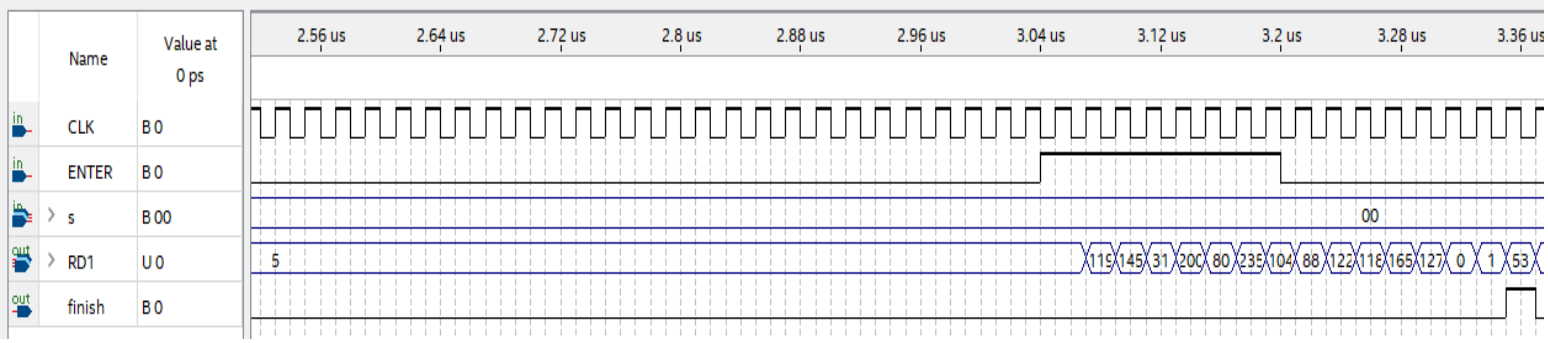
Read All Data:



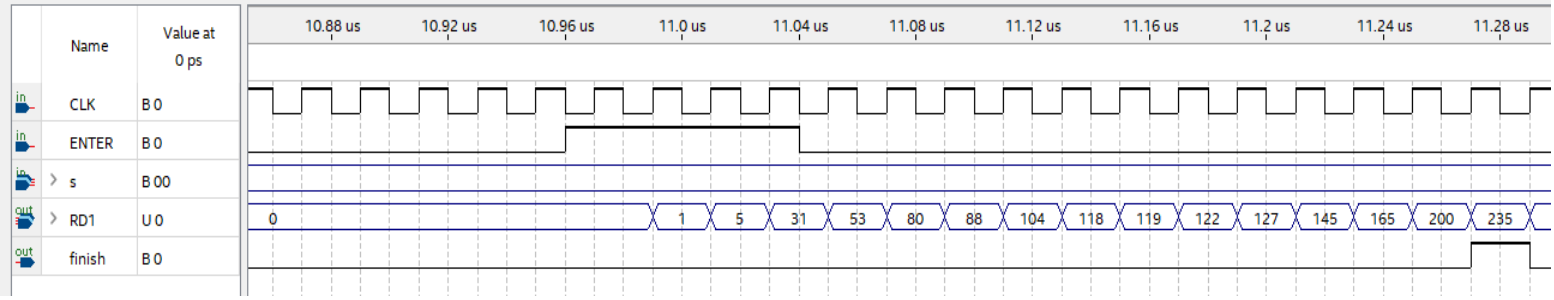
Waveform 2:



Read before sorting:



Read after sorting:



8- Conclusion:

To sum up, the design highlighted the crucial idea of separating each function in the design as a black box and dissertating the big parts into small manageable parts that may be realized using FSMs. This idea made the debugging process easier which in turn led to minimizing these bugs till the design became fully functional as verified by the simulation waveforms.