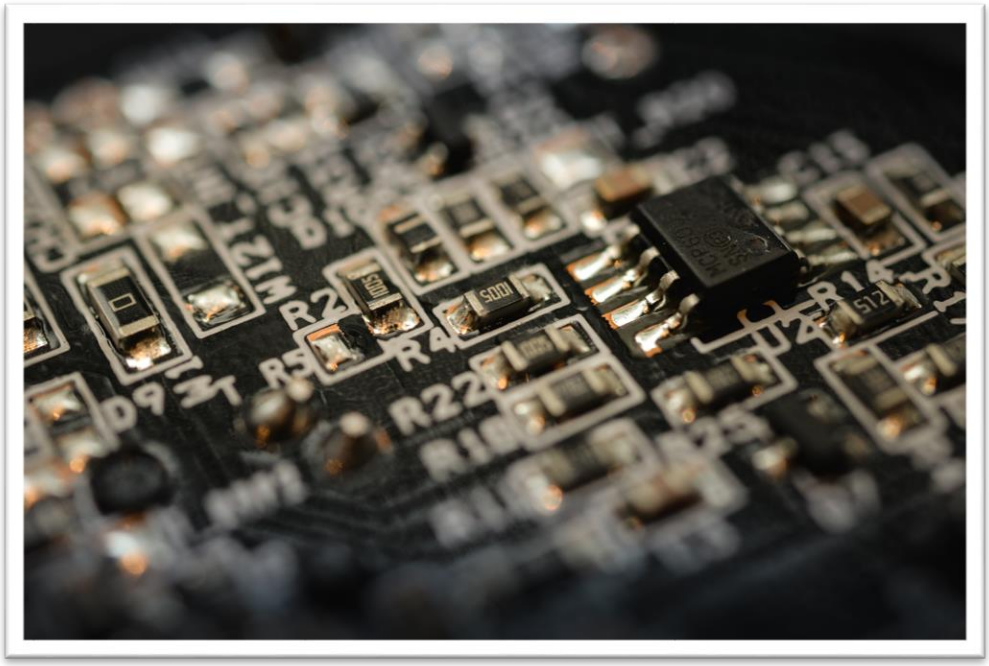


Sorting algorithm hardware accelerator

Electronics Systems

Riccardo Fiorini, Erica Raffa

Introduction

The Sorter is a circuit designed to optimise the performance of the sorting operation of the system. Let **'M'** be the maximum number of elements (symbols) that can be sorted and **'N'** the size in bit of the individual elements, both defined by the user. The circuit stores M elements in memory, sorting them in *increasing* order. The structure of the Sorter was designed as a **list** of components that store the new value, sorted on the way through.

The input symbols are stored and processed by the sorting algorithm only if the *write_enable* signal is high. The read procedure is activated when *read_symbol* is high, and it consists in reading all the elements from the 'head' of the list and forwarded by *symbol_out*.

Architecture

The architecture is designed with a Sorter entity composed by a certain number of Cell components, depending on the size M of the sorter. Every cell has a dimension of N bits, depending on the size of the symbols that must be sorted.

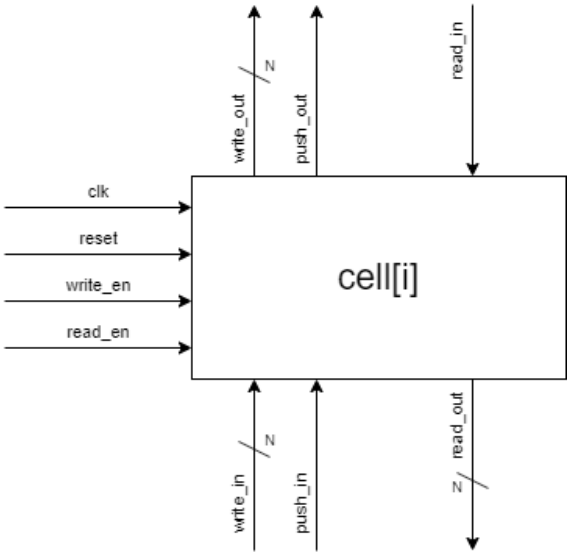
Cell entity

The Cell entity has the following ports:

- Input
  - *clk*: std_logic
  - *reset*: std_logic (asynchronous and active low)
  - *write_en*: std_logic (enable write operation)
  - *read_en*: std_logic (enable read operation)
  - *read_in*: std_logic_vector (size N, receive the symbol to be read)
  - *write_in*: std_logic_vector (size N, receive a new symbol to be sorted)
  - *push_in:* boolean (check whether there is a symbol incoming)
- Output
  - *read_out*: std_logic_vector (size N, forward the symbol to be read)
  - *write_out*: std_logic_vector (size N, forward a new symbol to be sorted)
  - *push_out:* boolean (to inform the next cell that a symbol is being pushed out)

And the following internal signals:

- Signals
  - *curr_data*: std_logic_vector (size N, stores the current data of a cell)
  - *occupied*: boolean (check whether the cell is storing a symbol or not)



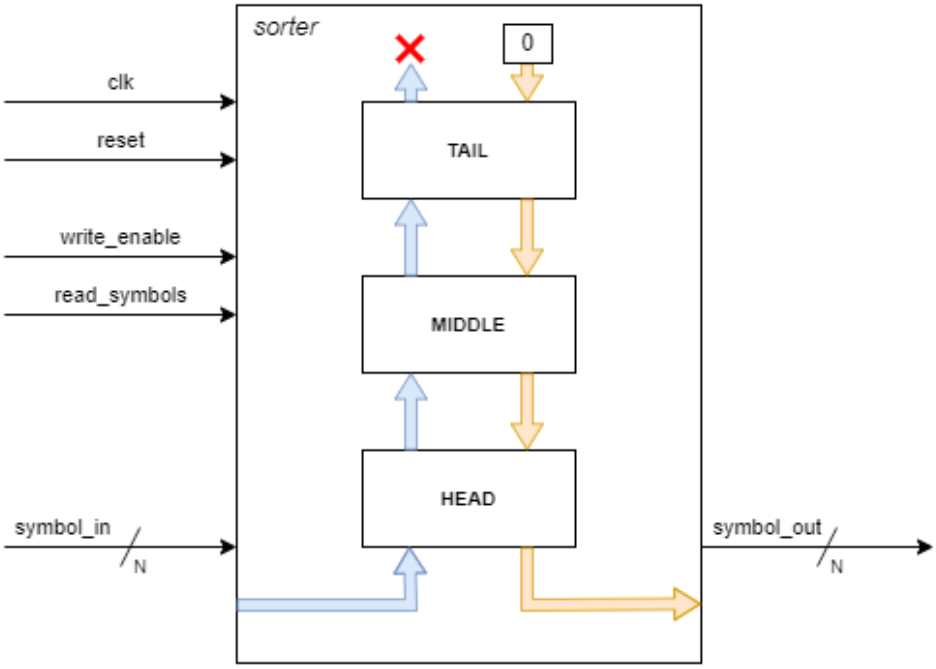
Sorter entity

The Sorter entity has the following ports:

- Input
  - *clk*: std_logic
  - *reset*: std_logic (asynchronous and active low)
  - *write_enable*: std_logic (enable write operation)
  - *read_symbols*: std_logic (enable read operation)
  - *symbol_in*: std_logic_vector (size N, receive the symbol to be sorted)
- Output
  - *symbol_out*: std_logic_vector (size N, output symbols that are being read)

And the following internal signals:

- Signals
  - *cell_data_write*: data_array (size M x N, array of std_logic_vector that forwards symbols between cells during writing operation)
  - *cell_data_read*: data_array (size M x N, array of std_logic_vector that forwards symbols between cells during reading operation)
  - *push_array*: bool_array (size M, array of boolean that forwards pushing info between cells during writing operation)



Read flow

When a cell is reading from an empty cell, its content is overwritten by the value x'00' and it becomes 'empty' after the current value is forwarded to the cell below. A similar behaviour occurs for the 'tail' cell, which always read the value 0 from above.

Moreover, the read operation has the priority over the write operation: when read_symbol is up, it does not matter if a new symbol has to be written.

Write flow

The write operation flow is piloted by the wires *push_in* and *push_out*. The signal forwarded by *push_out* notifies the cell above that a new symbol is been forwarded. The signal *push_out* of a cell[i] is linked to the *push_in* of the cell[i+1].

Test-plan

In our Testbench we decided to verify some borderline cases and possible error situations, explaining **why** certain scenarios occur and **how** we decided to handle them.

Borderline cases

Conflict of simultaneous reading and writing

In case of simultaneous reading and writing, the read operation is performed. We decided to prioritize the reading task because we consider it as the main goal of the sorter: a read request will be always fulfilled over a writing request. In the other way around, if the write request had the priority, we could never be able to read the content of the sorter in case of conflicts.

Overflow handling

In the case of the writing operation, if more than M symbols are given, they are sorted anyway and the greater $(M+1)^{th}$ symbol, between the new one and the stored ones, is discarded from the last cell.

Read request with empty sorter

In case of a read request while the sorter is empty, *symbol_out* is initialized anyway with the current data of the 'head' cell. Since an empty cell is initialized with x'00', the latter will be returned.

Possible error situations

Insertion of symbol '0'

The insertion of multiple symbols with value '0' becomes a problem if the next read operation is performed during a number of clock cycle lower than the number of consecutive 0s inserted. This happens because the reading of a '0' from the cell above resets the cell below, changing its *occupied* status to false.

During the following write operation the first cells are seen as not occupied, even if the symbol '0' is stored inside. As a result, the new symbol is inserted in the first not occupied cell, leaving the sorter in a non-consistent state.

Residual data during read operation

This happens every time a new read operation is requested. The symbols forwarded during the write operation, which are not yet stored in any cell, are deleted. In any case, the sorter will remain in a consistent state but it results in data loss.

Results

Synthesis on Zync Xilinx FPGA (with Vivado tool)

- Utilization

Name	Slice LUTs (17600)	Slice Registers (35200)	Bonded IOB (100)	BUFGCTRL (32)
sorter_wrapper	91	95	20	1
sorter_i (sorter)	91	95	0	0
GEN[0].FIRST.CELL_HEAD (cell)	19	26	0	0
GEN[1].MIDDLE.CELL_MID (cell_0)	36	26	0	0
GEN[2].MIDDLE.CELL_MID (cell_1)	22	26	0	0
GEN[3].LAST.CELL_TAIL (cell_2)	14	17	0	0

The number of registers is the same in each cell, except for the last one, where some ports are not mapped; the number of LUTs allocated to each cell decrease from one cell to the other.

- Timing

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	3.929 ns	Worst Hold Slack (WHS):	0.144 ns	Worst Pulse Width Slack (WPWS):	3.500 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	177	Total Number of Endpoints:	177	Total Number of Endpoints:	96

All user specified timing constraints are met.

The **Worst Negative Slack** is positive, so the timing requirement is respected: it means that the data arrives before the setup time of the capturing register, even in the critical path. The selected clock period is 8.000 ns, the **minimum clock period** is 4.071 ns, and the theoretical **maximum clock frequency** is 245 MHz.

Implementation on Zync Xilinx FPGA (with Vivado tool)

- Utilization

Name	Slice LUTs (17600)	Slice Registers (35200)	Slice (4400)	LUT as Logic (17600)	Bonded IOB (100)	BUFGCTRL (32)
sorter_wrapper	89	95	32	89	20	1
sorter_i (sorter)	89	95	32	89	0	0
GEN[0].FIRST.CELL_HEAD (cell)	18	26	11	18	0	0
GEN[1].MIDDLE.CELL_MID (cell_0)	35	26	20	35	0	0
GEN[2].MIDDLE.CELL_MID (cell_1)	22	26	17	22	0	0
GEN[3].LAST.CELL_TAIL (cell_2)	14	17	8	14	0	0

The utilization report after the implemented design is basically identical to the one of the synthesized designs.

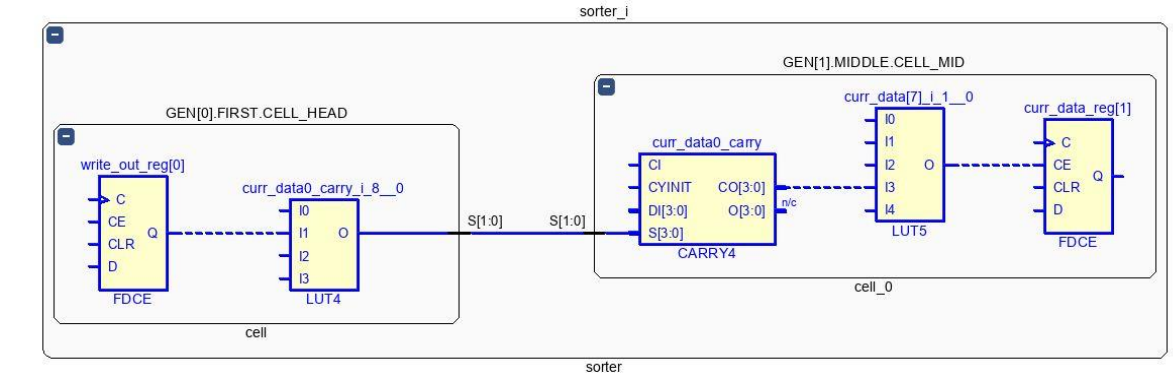
- Timing

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	3.695 ns	Worst Hold Slack (WHS):	0.155 ns	Worst Pulse Width Slack (WPWS):	3.500 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	177	Total Number of Endpoints:	177	Total Number of Endpoints:	96

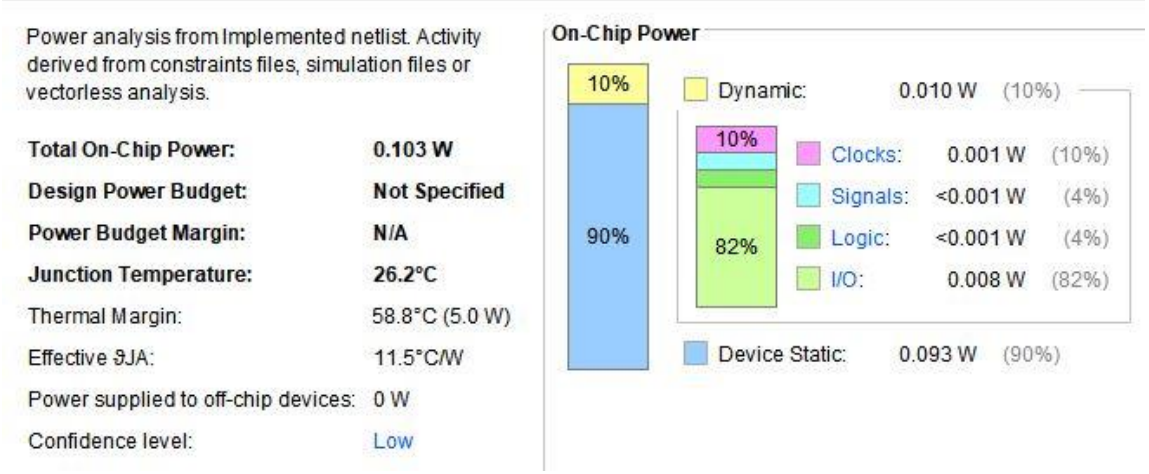
All user specified timing constraints are met.

Name	Slack	Levels	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement
Path 1	3.695	3	9	sorter_i/GEN[0]....ite_out_reg[0]/C	sorter_i/GEN[1]...data_reg[1]/CE	4.002	1.236	2.766	8.000
Path 2	3.695	3	9	sorter_i/GEN[0]....ite_out_reg[0]/C	sorter_i/GEN[1]...data_reg[3]/CE	4.002	1.236	2.766	8.000



The Timing Summary is similar to the synthesis one: the **WSN** is lower, but the time requirement is met still; the selected clock period is 8.000 ns, the **minimum clock period** is 4.305 ns, and the theoretical **maximum clock frequency** is 232 MHz.

- Power Consumption



The **static power consumption** depends on the utilization of the resources, while the **dynamic power consumption** depends on the switching activity of the signals (the default value is 50%).

Conclusions

The implementation of the Sorter, despite its simplicity, allows us to obtain satisfying results. Anyway, we propose the following ideas to solve some of the errors, that was discussed before, which can occur using our design.

NAND logic

The implementation of a NAND logic, comparing the counter *count_elem* (which counts the number of symbols inserted) with M as the maximum number of cells, would replace the constant value '1' in input of the *push_in* of the **head** cell. In this way, if the list is full, the new symbol is discarded.

Using counters

The insertion of a counter *count_to_read* inside every cell, allows the read operation to take out all the symbols stored inside the sorter since the cells know exactly how many symbols they have to read from above. This implementation resolves the "fake" empty cell problem, since the stored symbol '0' cannot be mistaken for an empty cell anymore.