A Compelling Case for Using BSV in Academia: Insights from Redesigning a Capstone Project

Govardhan | <Tentative> January, 2025

Introduction

At InCore, the use of Bluespec SystemVerilog (BSV) is one of the superpowers that enables small teams like ours to specify complex hardware intuitively, and efficiently.

Like most undergraduate students, I started out designing my initial digital logic in Verilog, with reference to coding guidelines from professors, coursework, and literature. Back then, specifying designs dealing with parallelism and concurrency in Verilog was always challenging due to its structural and timing emphasis on thinking "how" the hardware designs look like instead of "what" rules, conditions and invariants need to be specified to ensure correct behavior, leaving the scheduling to the Bluespec compiler.

For example, implementing a FIFO buffer:

- In Verilog, you’d think about the enqueue/dequeue timing, full/empty flags, and explicitly manage the read/write pointers.
- In BSV, you’d write rules like "when not full and enqueue requested, add element" and let the compiler handle the scheduling.

A year ago, as a novice BSV designer at InCore, I decided it would be meaningful to re-implement the Multi-Dimensional Sorting Algorithm (MDSA) implementation from my capstone project in a High-Level Hardware Description Language - BSV.

Before addressing the timing and area concerns of this redesign, the focus of this work is to evaluate if the BSV counterpart is cycle-level performant, intuitive to understand, and easy to implement.

More on the taxonomy of sorters, low power methodologies, other variants (Hybrid and Odd-Even sorters), and ASIC implementation results can be referred to in the published paper Low Power Multidimensional Sorters using Clock Gating and Index Sorting.

My complete MDSA Bitonic Implementation in BSV, along with the legacy Verilog implementation, can be found in my GitHub repository.

TLDR; In this blog, we shall:

1. Understand the Multi-Dimensional Sorting Algorithm,
2. Use inbuilt BSV Library functions to design an increasing complexity of sorters,
3. Sort 64 elements passed as an 8x8 array in six phases,

4. Discover a 25% reduction in gate-count (121k to 90k) by comparing the legacy Verilog implementation vs the new BSV implementation using a simple optimization.

The Compare And Exchange Block

The `Compare And Exchange` (CAE) block is a fundamental building block of Systolic Array based Parallel Hardware Sorters which sorts two inputs to an ascending order output.

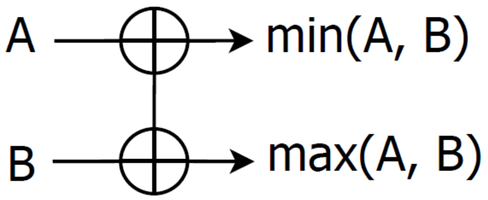

Figure 1. The Compare and Exchange (CAE) Block

- First, we specify the CAE typedef as a Vector of 2 elements of width `WordLength`:

```
typedef Vector#(2, Bit#(WordLength)) CAE;
```

- Then, declare the method ActionValue `mav_get_sort`:

The CAE block checks if `cae_in[0]` is greater than `cae_in[1]` to swap them.

TIP We can use the inbuilt Vector to Vector `reverse` function to swap the values.

```
method ActionValue#(CAE) mav_get_sort (CAE cae_in);
   if(cae_in[0] > cae_in[1]) begin
      cae_in = reverse(cae_in);
   end
   return(cae_in);
endmethod
```

If you're curious what the generated verilog for the CAE looks like, here's a snippet:

```
assign mav_get_sort =
        (mav_get_sort_cae_in[31:0] <= mav_get_sort_cae_in[63:32]) ?
        mav_get_sort_cae_in :
        { mav_get_sort_cae_in[31:0], mav_get_sort_cae_in[63:32] } ;
```

Here, the `mav_get_sort` output will only compare the two 32-bit inputs to the sorter, i.e. check if the first input `mav_get_sort_cae_in[31:0]` is less than or equal to the second input `mav_get_sort_cae_in[63:32]` and return the output as `mav_get_sort_cae_in`, else return the ascending order: `{ mav_get_sort_cae_in[31:0], mav_get_sort_cae_in[63:32] }`.

The Bitonic Sorting Unit

The Bitonic Sorting Unit is a network of 24 such CAE blocks, intricately arranged as depicted below.

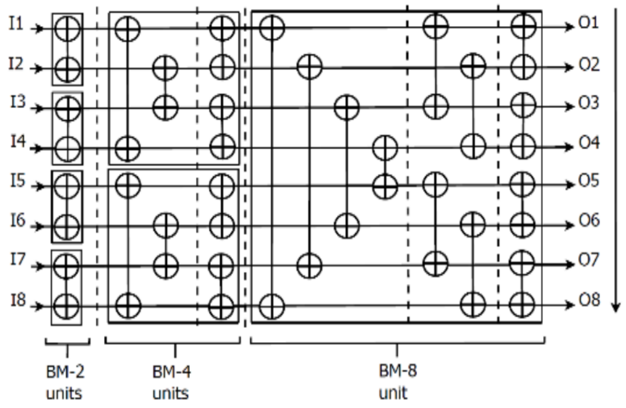

Figure 2. The Bitonic Merge Sorting Network

This network sorts eight input elements in ascending order at the end of six stages.

To read more about the Bitonic Sorting Network, refer to the seminal paper on systolic array sorting network design by Batcher[4].

If you look closely, we can take parts of the above BM8 architecture and modularize them.

The BM4 sorter

We could modularize a part of this design, the BM4 unit, by creating an intermediate two-stage, four-input sorter, and specify the two methods for input and output as follows:

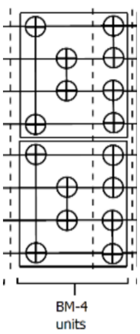

Figure 3. The BM4 sorter Network

- Declare a typedef for `BM4` as a vector of 4 inputs of width `WordLength`:

```
typedef Vector#(4, Bit#(WordLength)) BM4;
```

- Specify the intermediate pipeline as a register of the BM4 type:

```
   Reg#(BM4) pipe <- mkReg(unpack(0));
```

- First stage of sorting with the inputs, by routing the inputs at indices 0 and 3 to CAE-0, and 1 and 2 to CAE-1 block.

TIP We use the inbuilt vector function `vec` to combine multiple elements into a vector:

```
      let lv_get_sort_1 <- cae[0].mav_get_sort(vec(bm4[0], bm4[3]));
      let lv_get_sort_2 <- cae[1].mav_get_sort(vec(bm4[1], bm4[2]));
      // Store intermediate results in a pipeline
      pipe <= vec(lv_get_sort_1[0], lv_get_sort_2[0], lv_get_sort_2[1],
lv_get_sort_1[1]);
```

- Perform the second stage sorting with the intermediate sorted values by routing the pipeline outputs at indices 0 and 1 to CAE-0, and 2 and 3 to CAE-1 block:

```
      let lv_get_sort_3 <- cae[0].mav_get_sort(vec(pipe[0], pipe[1]));
      let lv_get_sort_4 <- cae[1].mav_get_sort(vec(pipe[2], pipe[3]));
```

- Return the outputs as:

```
      return (vec(lv_get_sort_3[0], lv_get_sort_3[1], lv_get_sort_4[0],
lv_get_sort_4[1]));
```

The BM8 sorter

Now with the abstraction of using a BM4 sorter, we can proceed to design the complete Bitonic Merge 8 input sorter as follows:

- Instantiate the 5 intermediate register pipelines:

```
    Vector#(5, Reg#(BM8)) pipe <- replicateM(mkReg(unpack(0)));
```

- Pass the inputs through the sorting network defined for each stage of the BM8, while storing the intermediate values in the above pipeline registers:
- Stage 1:

```
method Action ma_get_inputs (BM8 bm8_in) if (rg_stage == INIT);

    let lv_cae_sort_1 <- cae_stage_1[0].mav_get_sort(vec(bm8_in[0], bm8_in[1]));
    let lv_cae_sort_2 <- cae_stage_1[1].mav_get_sort(vec(bm8_in[2], bm8_in[3]));
    let lv_cae_sort_3 <- cae_stage_1[2].mav_get_sort(vec(bm8_in[4], bm8_in[5]));
    let lv_cae_sort_4 <- cae_stage_1[3].mav_get_sort(vec(bm8_in[6], bm8_in[7]));
```

```
pipe[0] <= vec(lv_cae_sort_1[0]
```

```
                  , lv_cae_sort_1[1]
                  , lv_cae_sort_2[0]
                  , lv_cae_sort_2[1]
                  , lv_cae_sort_3[0]
                  , lv_cae_sort_3[1]
                  , lv_cae_sort_4[0]
                  , lv_cae_sort_4[1]);
```

- Stage 2:

Pass the outputs of the first stage to the BM4 sorter, and register their output for the third stage:

```
    bm4_stage_2_3[0].ma_get_inputs(vec(pipe[0][0], pipe[0][1], pipe[0][2],
pipe[0][3]));
    bm4_stage_2_3[1].ma_get_inputs(vec(pipe[0][4], pipe[0][5], pipe[0][6],
pipe[0][7]));
```

```
    pipe[1] <= vec(lv_get_bm4_sort_1[0]
                  , lv_get_bm4_sort_1[1]
                  , lv_get_bm4_sort_1[2]
                  , lv_get_bm4_sort_1[3]
                  , lv_get_bm4_sort_2[0]
                  , lv_get_bm4_sort_2[1]
                  , lv_get_bm4_sort_2[2]
                  , lv_get_bm4_sort_2[3]);
```

... and so on for the remaining stages 4 to 6.

The MDSA Implementation

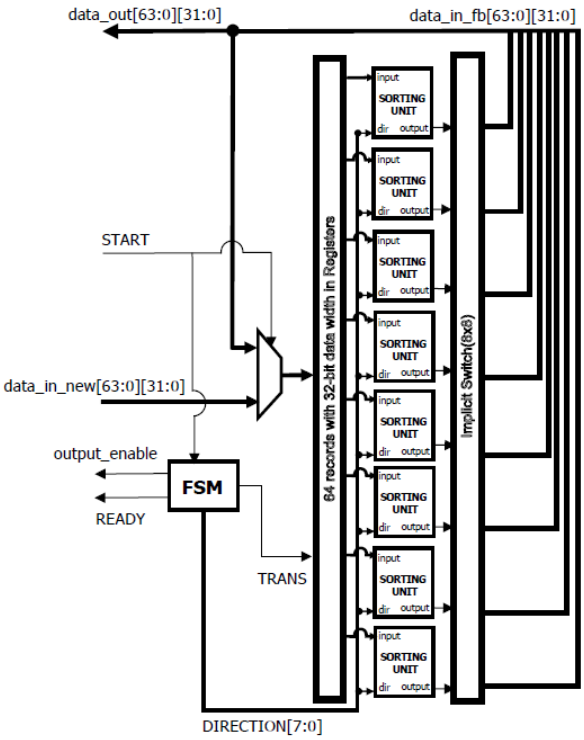

Figure 4. The block diagram of the MDSA Architecture

The MDSA's algorithm efficiently uses Parallel Hardware Sorter Arrays(PHSAs) like the Bitonic sorter we earlier designed to specify an architecture that uses eight such units to sort 64 elements in 6 stages by simply alternating between row and column sorting, and rerouting the order of outputs (ascending/descending).

The MDSA FSM

Phase	Row/ Column Sorting	Ascending sorting	Descending sorting
1	Column	1,2,3,4,5,6,7,8	-
2	Row	1,3,5,7	2,4,6,8
3	Column	1,2,3,4,5,6,7,8	-
4	Row	2,4,6,8	1,3,5,7
5	Column	1,2,3,4,5,6,7,8	-
6	Row	1,2,3,4,5,6,7,8	-

Figure 5. The FSM that implements the MDS-Algorithm

Also, now that we are redesigning the MDSA, I desire to implement an optimization where I handle the ascending and descending order of the MDSA inputs at the input of the Parallel Sorting Networks and limit myself to use unidirectional CAE blocks.

In the legacy verilog implementation, we had CAE blocks that could support both the directions, and kept the top level simple such that we only transpose between row and column sorting between phases.

IMPORTANT	The reason why I choose to keep the CAEs simple is because selecting between the ascending and descending increases the number of multiplexors in each CAE block, and in the MDSA sorter, there are 8 (Bitonic Networks) * 24 (CAE blocks in each Network) = 192 CAE blocks which is indeed substantial! Let us later look at the gate-count difference to

> determine how much this optimization has impacted our implementation's area :)

We specify the `MDSA_64` type which is a multidimensional 8x8 vector

```
typedef Vector#(8, Vector#(8, Bit#(WordLength))) MDSA_64;
```

To create a 64 record register buffer specified as:

```
    Reg#(MDSA_64) v_rg_mdsa_in <- mkReg(unpack(0));
```

And use this helper function to send inputs to the MDSA sorter network:

```
    function fn_input_sorting_network(Vector#(8, Ifc_bm8) bm8
                                    , MDSA_64 mdsa_in);
        action
            bm8[0].ma_get_inputs(mdsa_in[0]);
            bm8[1].ma_get_inputs(mdsa_in[1]);
            bm8[2].ma_get_inputs(mdsa_in[2]);
            bm8[3].ma_get_inputs(mdsa_in[3]);
            bm8[4].ma_get_inputs(mdsa_in[4]);
            bm8[5].ma_get_inputs(mdsa_in[5]);
            bm8[6].ma_get_inputs(mdsa_in[6]);
            bm8[7].ma_get_inputs(mdsa_in[7]);
        endaction
```

Stage 1: Column Sorting

- Sending the inputs to the Eight BM8 sorters:

Phase	Row/ Column Sorting	Ascending sorting	Descending sorting
1	Column	1,2,3,4,5,6,7,8	-
2	Row	1,3,5,7	2,4,6,8
3	Column	1,2,3,4,5,6,7,8	-
4	Row	2,4,6,8	1,3,5,7
5	Column	1,2,3,4,5,6,7,8	-
6	Row	1,2,3,4,5,6,7,8	-

Figure 6. The First Phase of the MDSA

```
    /*------------------ STAGE 1 ----------------*/
    rule rl_mdsa_send_inputs_to_stage_1(rg_mdsa_fsm == STAGE_1_IN);
        // Column Sorting Phase
        fn_display($format("[MDSA] STARTING MDSA STAGE 1"));
        fn_display($format("[MDSA]: STAGE 1 INPUTS:", fshow(v_rg_mdsa_in)));
        fn_input_sorting_network(bm8, v_rg_mdsa_in);
        rg_mdsa_fsm <= STAGE_1_OUT;
```

- Collecting the ascending order of responses

```
        // Ascending (0,1,2,3,4,5,6,7)
        lv_s1_output[0] <- bm8[0].mav_return_outputs();
        lv_s1_output[1] <- bm8[1].mav_return_outputs();
        lv_s1_output[2] <- bm8[2].mav_return_outputs();
        lv_s1_output[3] <- bm8[3].mav_return_outputs();
        lv_s1_output[4] <- bm8[4].mav_return_outputs();
        lv_s1_output[5] <- bm8[5].mav_return_outputs();
        lv_s1_output[6] <- bm8[6].mav_return_outputs();
```

- Transposing the output from Column to Row sorting:

TIP We can use the inbuilt Vector to Vector `transpose` function in BSV to alternate between the row and column sorting between the phases of the MDSA.

```
        v_rg_mdsa_in <= transpose(lv_s1_output);
```

Stage 2: Row Sorting

- Sending the inputs to the Eight BM8 sorters:

Phase	Row/ Column Sorting	Ascending sorting	Descending sorting
1	Column	1,2,3,4,5,6,7,8	-
2	Row	1,3,5,7	2,4,6,8
3	Column	1,2,3,4,5,6,7,8	-
4	Row	2,4,6,8	1,3,5,7
5	Column	1,2,3,4,5,6,7,8	-
6	Row	1,2,3,4,5,6,7,8	-

Figure 7. The Second Phase of the MDSA

```
    rule rl_mdsa_send_inputs_to_stage_2(rg_mdsa_fsm == STAGE_2_IN);
        // Row Sorting Phase
        fn_display($format("[MDSA] STARTING MDSA STAGE 2"));
        fn_display($format("[MDSA]: STAGE 2 INPUTS:", fshow(v_rg_mdsa_in)));
        fn_input_sorting_network(bm8, v_rg_mdsa_in);
        rg_mdsa_fsm <= STAGE_2_OUT;
```

- Collecting the alternating ascending and descending order of responses

```
        lv_s2_output[0] <- bm8[0].mav_return_outputs();
        lv_s2_output[1] <- bm8[1].mav_return_outputs();
        lv_s2_output[1] = reverse(lv_s2_output[1]);
        lv_s2_output[2] <- bm8[2].mav_return_outputs();
        lv_s2_output[3] <- bm8[3].mav_return_outputs();
        lv_s2_output[3] = reverse(lv_s2_output[3]);
        lv_s2_output[4] <- bm8[4].mav_return_outputs();
```

```
        lv_s2_output[5] <- bm8[5].mav_return_outputs();
        lv_s2_output[5] = reverse(lv_s2_output[5]);
        lv_s2_output[6] <- bm8[6].mav_return_outputs();
        lv_s2_output[7] <- bm8[7].mav_return_outputs();
```

- Transposing the output from Row to Column sorting:

```
        v_rg_mdsa_in <= transpose(lv_s2_output);
```

... and so on for the remaining stages 3 to 6 as per the MDSA FSM.

Ultimately, an ideal test case where all 64 inputs specified in descending order is:

```
[MDSA] STARTING MDSA STAGE 1
[MDSA]: STAGE 1 INPUTS:<V <V 'h00000040 'h0000003f 'h0000003e 'h0000003d 'h0000003c
'h0000003b 'h0000003a 'h00000039  > <V 'h00000038 'h00000037 'h00000036 'h00000035
'h00000034 'h00000033 'h00000032 'h00000031  > <V 'h00000030 'h0000002f 'h0000002e
'h0000002d 'h0000002c 'h0000002b 'h0000002a 'h00000029  > <V 'h00000028 'h00000027
'h00000026 'h00000025 'h00000024 'h00000023 'h00000022 'h00000021  > <V 'h00000020
'h0000001f 'h0000001e 'h0000001d 'h0000001c 'h0000001b 'h0000001a 'h00000019  > <V
'h00000018 'h00000017 'h00000016 'h00000015 'h00000014 'h00000013 'h00000012
'h00000011  > <V 'h00000010 'h0000000f 'h0000000e 'h0000000d 'h0000000c 'h0000000b
'h0000000a 'h00000009  > <V 'h00000008 'h00000007 'h00000006 'h00000005 'h00000004
'h00000003 'h00000002 'h00000001  >  >
```

Shall be sorted in 6 stages to ascending order as follows:

```
Final MDSA output: <%h><V <V 'h00000001 'h00000002 'h00000003 'h00000004 'h00000009
'h0000000a 'h0000000b 'h0000000c  > <V 'h00000005 'h00000006 'h00000007 'h00000008
'h0000000d 'h0000000e 'h0000000f 'h00000010  > <V 'h00000011 'h00000012 'h00000013
'h00000014 'h00000019 'h0000001a 'h0000001b 'h0000001c  > <V 'h00000015 'h00000016
'h00000017 'h00000018 'h0000001d 'h0000001e 'h0000001f 'h00000020  > <V 'h00000021
'h00000022 'h00000023 'h00000024 'h00000029 'h0000002a 'h0000002b 'h0000002c  > <V
'h00000025 'h00000026 'h00000027 'h00000028 'h0000002d 'h0000002e 'h0000002f
'h00000030  > <V 'h00000031 'h00000032 'h00000033 'h00000034 'h00000039 'h0000003a
'h0000003b 'h0000003c  > <V 'h00000035 'h00000036 'h00000037 'h00000038 'h0000003d
'h0000003e 'h0000003f 'h00000040  >  >
Verilog simulation finished
```

Steps to generate verilog and run simulations of the CAE, BM4, BM8 and MDSA_Bitonic in the GitHub Repository

1. Clone the repository:

```
git clone https://github.com/govardhnn/Low_Power_Multidimensional_Sorters.git
```

2. Navigate to the build directory of the BSV collateral

```
cd bsv/build
```

3. Modify the `makefile.inc` to select the module to simulate

For CAE/BM4/BM8/MDSA:

```
TB_BSV:= cae_testbench / bm4_testbench / bm8_testbench / mdsa_bitonic_testbench
```

1. To run the simulation with the Bluespec Compiler (bsc):

```
make all_vsim
```

The generated verilog can be found in the `verilog_dir` directory

NOTE	To get a vcd dump of the simulation, rerun the executable with the +bscvcd argument. E.g. ./mk_mdsa_bitonic_testbench_vsim +bscvcd Or, add the +bscvcd flag to the `v_simulate` target in the Makefile

Observations

- Does this new design meet the timing specification?

Given each clock cycle is 10ns (Freq: 100MHz) for this simulation,

Inputs provided to the MDSA at [15ns]

```
[15][MDSA] STARTING MDSA STAGE 1
[15][MDSA]: STAGE 1 INPUTS:<V <V 'h00000040 'h0000003f ... 'h00000002 'h00000001  >  >
```

takes 6 cycles → 60ns more to complete sorting at [75ns]

```
[75][MDSA]: STAGE 1 DONE
[75][MDSA]: STAGE 1 OUTPUTS:<V <V 'h00000039 'h0000003a ... 'h00000007 'h00000008  >
>
```

Taking a dedicated 10ns for the transpose matrix, the complete operation of sorting 64 numbers would take: (70ns x 6) 420ns from the start of the simulation(15ns) which is 435ns.

```
[435][MDSA] DONE. Returning outputs:<V <V 'h00000001 'h00000002 ... 'h0000003f
'h00000040  >  >
```

Hence, we have proven the timing specification level correctness of this design, let us compare the line-count and gate-count efficiencies.

- How many lines of BSV did this take?

In the BSV implementation of the MDSA

```
 46 cae.bsv
 87 bm4.bsv
153 bm8.bsv
246 mdsa_bitonic.bsv
```

Total lines of BSV written: 532

Which generates the following verilog modules

```
66  mk_cae.v
171 mk_bm4.v
575 mk_bm8.v
829 mk_mdsa_bitonic.v
```

Total lines of Verilog generated: 1641

TIP You can disable the $display statements from being generated in the verilog generation phase by removing the define `-D DISPLAY` in the Makefile

Finally, the Gate-Count Comparison of the legacy Verilog MDSA vs the new BSV MDSA implementations:

NOTE We shall be using the open-source synthesis tool Yosys, and the scripts can be found in `bsv/build/synth.ys` and `verilog/MDSA_bitonic/synth.ys`

Legacy Verilog MDSA synthesis

To run synth of the legacy Verilog MDSA codebase with top module `MDSA_top`:

```
cd verilog/MDSA_bitonic
```

and run:

```
yosys -s synth.ys
```

And we shall find that the synthesis gate count is:

```
Number of wires:              143051
Number of wire bits:          247433
Number of public wires:         1674
Number of public wire bits:    56173
Number of memories:                0
Number of memory bits:             0
Number of processes:               0
Number of cells:              121574
```

BSV MDSA synthesis

NOTE We shall be running synthesis on the verilog generated from the BSV explained in this blog.

You can run a yosys synth at the `bsv/build` with top module `mk_mdsa_bitonic`

```
cd bsv/build
make yosys_synth
```

And the synthesis gate count is:

```
Number of wires:              108735
Number of wire bits:          243796
Number of public wires:         2576
Number of public wire bits:   101182
Number of memories:                0
Number of memory bits:             0
Number of processes:               0
Number of cells:               90410
```

*There you go! A reduction in gate-count from **121k** to **90k**.* A staggering **25%** reduction in the total number of cells.

Scope of future contribution to this project

- Implementing the Hybrid and Odd Even Sorter designs in BSV - which is pretty fun and replicable looking at the current BSV implementation of the Bitonic Sorter and referring to papers [4] and [6].
- Deep-dive into the logic that changed between the legacy Verilog MDSA and the new BSV redesign of the MDSA.

- Implement Index Sorting for the Sorting Networks in BSV referring to [1] and [7].
- Understand the generated verilog from each module in BSV, and try to find scope for optimizing area and timing.

Acknowledgements

Many thanks to the peers at InCore who provided valuable feedback on early drafts of this blog.

The block diagrams and drawings to aid the explanation of the CAE, Bitonic and MDSA are from the paper[1], and the legacy Verilog codebase from the team Samahith S A, Manogna R, Hitesh D including myself, guided by our Professor Dr. Sudeendra Kumar at PES University, Bangalore.

Initial Work on MDSA[5] and PHSA sorter variants[6] have been the guiding light for the theory that has led to the innovation in the field of hardware sorting.

The BSV language compiler [2] and libraries reference guide from Bluespec(inc) are the resources that have helped realize the superpower(BSV) that this blog is based upon.

References

1. "Low Power Multidimensional Sorters using Clock Gating and Index Sorting." 2023 IEEE International Conference on Electronics, Computing and Communication Technologies (CONECCT)
2. The Bluespec Compiler(bsc)
3. "Bluespec Libraries Reference Guide"
4. "Sorting Networks and Their Applications," Proc. AFIPS Proc. Spring Joint Computer Conf., 1968.
5. "RTHS: A Low-Cost High-Performance Real-Time Hardware Sorter, Using a Multidimensional Sorting Algorithm" in IEEE Transactions on Very Large-Scale Integration (VLSI) Systems, July 2019.
6. "Design of Hybrid Sorting Unit," 2019 International Conference on Smart Structures and Systems (ICSSS), Chennai, India, 2019
7. "Index and Sort Algorithm Based on FPGA for Sorting Data," 2021 9th International Japan-Africa Conference on Electronics, Communications, and Computations (JAC-ECC).