Why I rewrote my capstone project in Bluespec SystemVerilog

Author: Sai Govardhan M C
Email: sai.govardhan@incoresemi.com

Introduction

This document outlines the motivation for using Bluespec SystemVerilog, which provides high levels of abstraction to rapidly design hardware microarchitecture.

Like most undergraduate students, I started out designing initial digital logic in Verilog, with reference to coding guidelines from professors, coursework, and the sunburst papers. Back then, specifying designs dealing with parallelism and concurrency was always a challenge due to lower levels of abstraction and regular rework to fix synthesis-simulation mismatches.

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

A year ago, as a novice BSV designer, I decided it would be interesting and meaningful to contrast the efforts that my team and I spent during our capstone project by re-implementing the Multi-Dimensional Sorting Algorithm (MDSA) in a High-Level HDL - BSV. As an outcome, this blog collates these insights to establish a strong case for teaching and using BSV at universities.

In this post, I shall be diving into my BSV implementation by explaining essential parts of the microarchitecture, the ease of specifying them in BSV, and corresponding snippets from the codebase.

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

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

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.

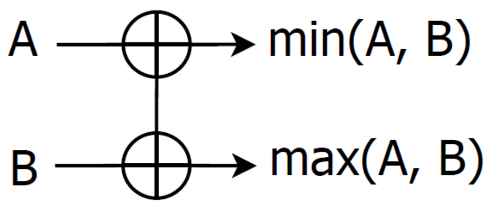


Figure 1. The Compare and Exchange (CAE) Block

- Specify the CAE typedef as a Vector of 2 elements of width `WordLength`:

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

- 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
```

The Bitonic Sorting Unit

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

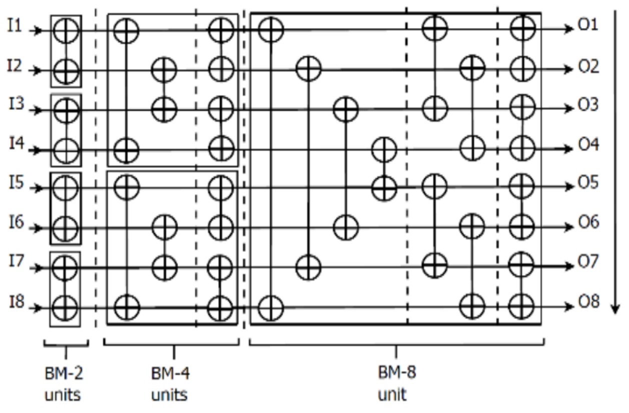


Figure 2. The Bitonic Merge Sorting Network

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:

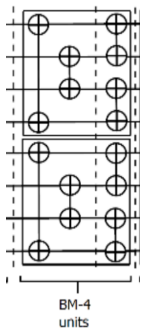


Figure 3. The BM4 sorter Network

- 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 Algorithm Implementation

The MDSA algorithm efficiently uses Parallel Hardware Sorters (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 only alternating between row and column sorting, and rerouting the order of outputs (ascending/descending).

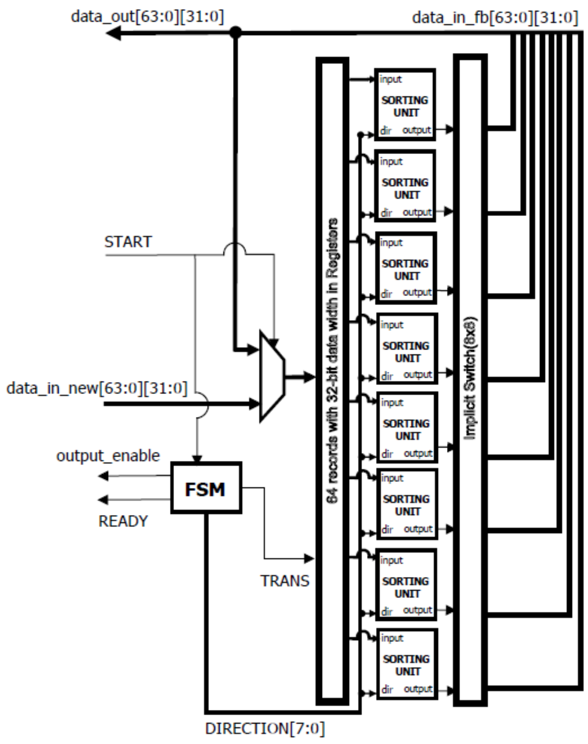
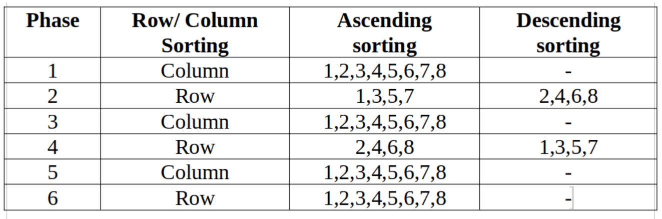


Figure 4. The block diagram of the MDSA Architecture

MDSA Agorithm 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

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:

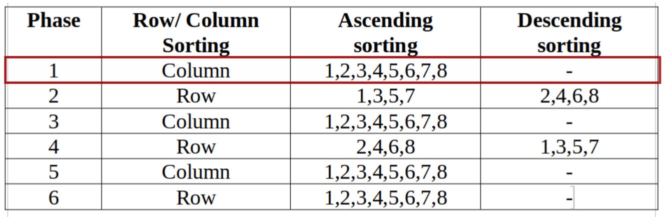
```
    // and is reused after each "phase" of sorting
```

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 Algorithm

```
    /*------------------ 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();
```

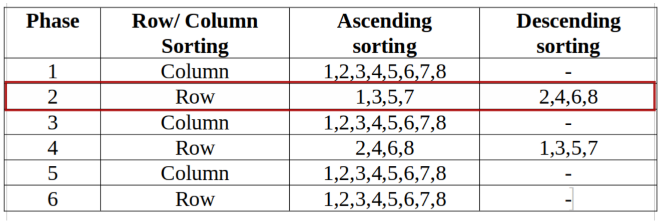
- Transposing the output:

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.

```
        // Transpose from Column Sorting to Row Sorting
```

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 Algorithm

```
    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

```
        // Transpose from Row Sorting to Column Sorting
```

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

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

```
[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 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 the the build directory of the BSV collateral

```
cd bsv/build
```

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

For CAE:

```
TB_BSV:= cae_testbench
```

For BM4:

```
TB_BSV:= bm4_testbench
```

For BM8:

```
TB_BSV:= bm8_testbench
```

For MDSA:

```
TB_BSV:= 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. Eg. ./mk_mdsa_bitonic_testbench_vsim +bscvcd Or, add the +bscvcd flag to the `v_simulate` target in the Makefile

Observations

1. Number of lines of code

In the BSV implementation of the MDSA

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

Which generates the following verilog modules

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

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

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`

```
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 number of cells - using a language much abstract, intuitive and efficient.**

Scope of contribution to this project(TODO)

Acknowledgements(TODO)

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 guided by my UG Professor Dr. Sudeendra Kumar at PES University, Bangalore.

The BSV languare compiler [2] and libraries reference guide from Bluespec(inc)

Initial Work on MDSA [5] and PHSA sorter variants[6]

<reviewers, thanks>

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. Libraries Reference Guide
4. K.E. Batcher, “Sorting Networks and Their Applications,” Proc. AFIPS Proc. Spring Joint Computer Conf., pp. 307-314, 1968.
5. A. Norollah, et al, "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, vol. 27, no. 7, pp. 1601-1613, July 2019.
6. V. S. Harshini et al, "Design of Hybrid Sorting Unit," 2019 International Conference on Smart Structures and Systems (ICSSS), Chennai, India, 2019, pp. 1-6