

Why I rewrote my capstone project in BSV

Author: Sai Govardhan M C

Email: sai.govardhan@incoresemi.com

Introduction

This document outlines the motivation to using Bluespec System Verilog which provides high levels of abstraction to rapidly design hardware microarchitecture.

As it happens with most Electronics students, I started out designing digital design logic, in Verilog, with reference to coding guidelines from the [Sunburst Papers](#) and careful guidance from my UG professor and course textbooks.

Specifying designs with parallelism and concurrency in Verilog 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 enabled a small teams like us to specify complex hardware intuitively, correctly and efficiently.

During my internship at InCore in 2024, as a novice BSV designer, I decided it would be meaningful to contrast the efforts that my team and I spent at college implementing the Multi Dimensional Sorting Algorithm(MDSA) in Verilog - by rewriting the MDSA Bitonic variant in BSV. This blog collates these insights, to establish a strong use-case of BSV at universities.

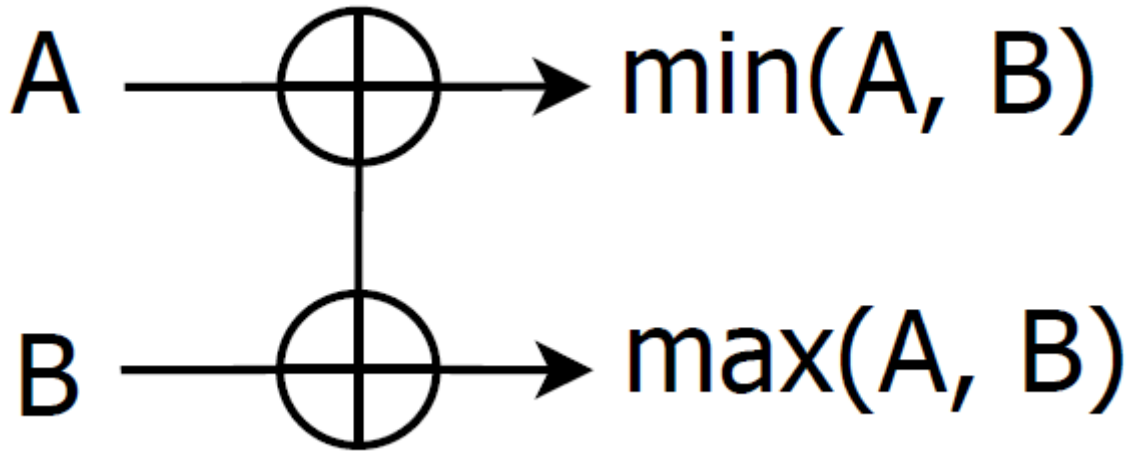
I shall be diving into the BSV implementation by explaining essential parts of the micro-architecture, and corresponding snippets from the code base.

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 hardware sorters. It simply compares two inputs and presents an ascending order output.



The CAE Block

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

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

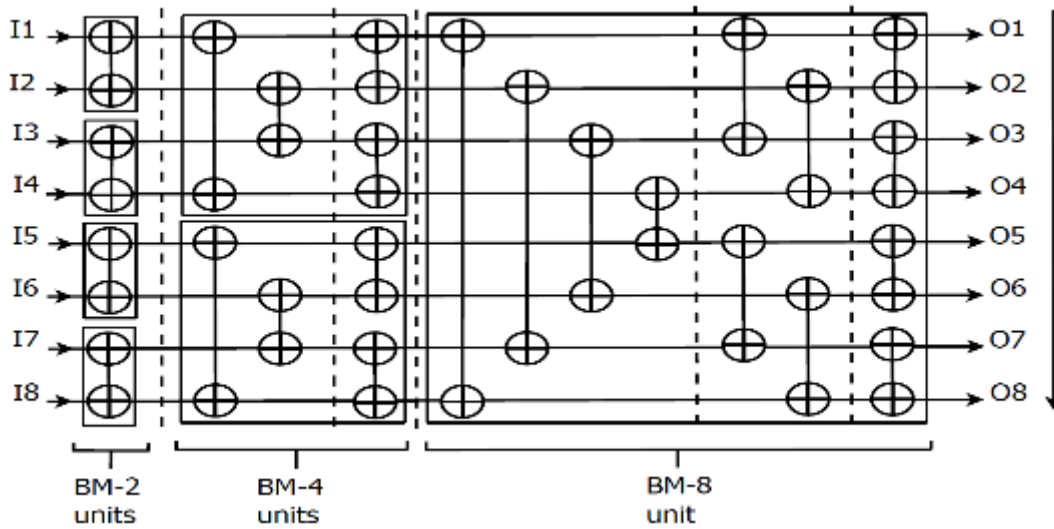
- Declare the method ActionValue 'mav_get_sort':

The CAE block checks if `cae_in[0]` is greater than `cae_in[1]` and uses the 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, arranged as depicted below. This network sorts 8 input elements in ascending order at the end of six stages.



The Bitonic Sorting Network

The BM4 sorter

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

- We specify the intermediate pipeline as:

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

- Perform the first stage sorting with the inputs, by routing the inputs at index 0 and 3 to the CAE-0, and 1 and 2 to CAE-1 block:

```
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 index 0 and 1 to the 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]));
```

- And 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 sorter as follows:

- Instantiate the 5 intermediate registers pipelines:

```
$display("[BM8]:[PIPE] Stage 1:", fshow(pipe[0]));
```

- Pass the inputs the the sorter and pass them through the network defined for each stage, while storing the intermediate values in the pipeline registers:
- Stage 1:

```
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:

```
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]));
```

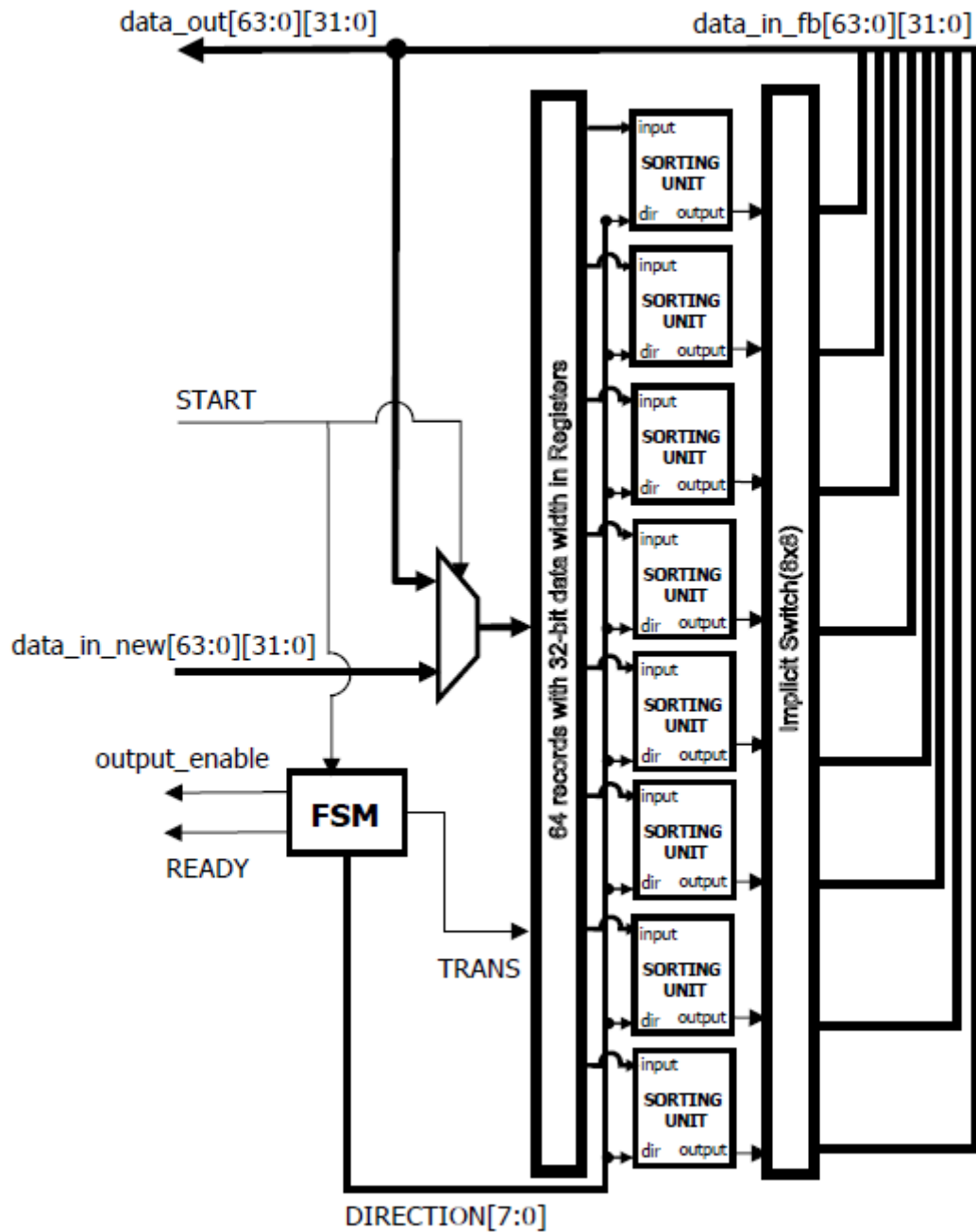
Register the outputs of the BM4 sorters for the third stage:

```
, 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.

The MDSA Algorithm Implementation

The MDSA algorithm stands out in its use of Parallel Hardware Sorters(PHSAs) like the Bitonic sorter we earlier designed to specify an architecture that uses 8 such units to sort 64 elements in 6 stages by alternating between row and column sorting, and rerouting the order of outputs(ascending/descending).



The block diagram of the MDSA Architecture

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	-

The FSM that implements the MDS-Algorithm

We specify the MDSA_64 type

```
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
endfunction
```

To create the Stage 1: Column Sorting

- Sending the inputs to the Eight BM8 sorters:

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

- Collecting the ascending order of responses

```
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();
lv_s1_output[7] <- bm8[7].mav_return_outputs();
```

Transposing the output

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

Stage 2: Row Sorting

- Sending the inputs to the Eight BM8 sorters:

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

- 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();
lv_s2_output[7] = reverse(lv_s2_output[7]);
```

Transposing the output

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

/... and so on for the remaining stages.

Ultimately, an ideal 64 numbers all in descending order sent as input:

```
[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:

```
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
```

References

Acknowledgements