RISC-V Student Competition CVA6 Optimization

Technical Report
Team: RISCy Business

Authors:
LISBOA, Felipe
SUZAWAKA, Henry

Email:
flisboa@telecom-paris.fr
henry.suzawaka@telecom-paris.fr

April 23, 2021

Contents

1 Introduction

This work aims at documenting the work performed in the context of the *1st national RISC-V student contest*, sponsored by **Thales**, the **GDR SoC2** and the **CNFM**. The competition consists in optimizing the ARIANE (CVA6) core [2] for FPGA targets.

ARIANE was primarily designed for ASIC targets, i.e ICs that are synthesized over standard cells. Although ARIANE can run on FPGA targets, the design can still benefit enormously from smart architectural improvements that enhance performance.

The remaining of this document is organized in the following structure:

Section 2 presents each one of the modifications we did in the code. First, the theoretical insight of each modification is presented. Second, we highlight key changes in the source code. Then, we proceed to show the results of that modification alone in terms of resource utilization, frequency gain and core mark results;

Section 3 discusses the impact of the modifications described in Section 2 all together and the final results of the fully modified core;

Section 4 mentions other modifications we tried but could not finish due to diverse reasons;

Section 5 concludes the work by discussing the overall optimization results and giving further optimization ideas.

Timing, utilization and performance reports proving the results presented in this document were provided to the organization, as required.

2 Proposed Optimisations

We provide one git branch for each optimisation proposed. The optimal way to read this document is to simultaneously look at the compare view of the indicate branch. In Section 3 we mention branches that assembles all the results.

2.1 Rewriting the register file code to use different FPGA memories

When it comes to porting ASIC designs to FPGA (or the contrary), memory synthesis is one of the main bottlenecks. One well-known difference involves the pre-existence of block RAMs on FPGAs versus synthesizing custom-sized hard-core RAMs on ASICs [1]. For the two optimizations described in the following, we substitute the custom-sized ASIC memory (which is synthesized as flip-flops in the FPGA) for the two pre-exiting types of FPGA memory: distributed RAMs (LUTRAM), which are synthesized using LUTs and spatially distributed across the FPGA matrix and block RAMs (BRAM), which are synchronous DRAM memory with fixed placement on the board.

2.1.1 Synthesizing with BRAM

GitHub branch: `regfile_tdp_bram`.

One of the constraints of the register file is the number of read/write ports. This constraint limits the ways that we could refactor the code, specially when the objective is to replace such module for one that uses FPGA's blocks. Such blocks have fixed number of ports and can, at most, operate on two different addresses (True Dual Port configuration, also known as TDP). In the current implementation of the code, the total number of addresses used to control the register file is four: two independent addresses for the write operation and other two addresses for the read operation, independent and possibly different from the write operation.

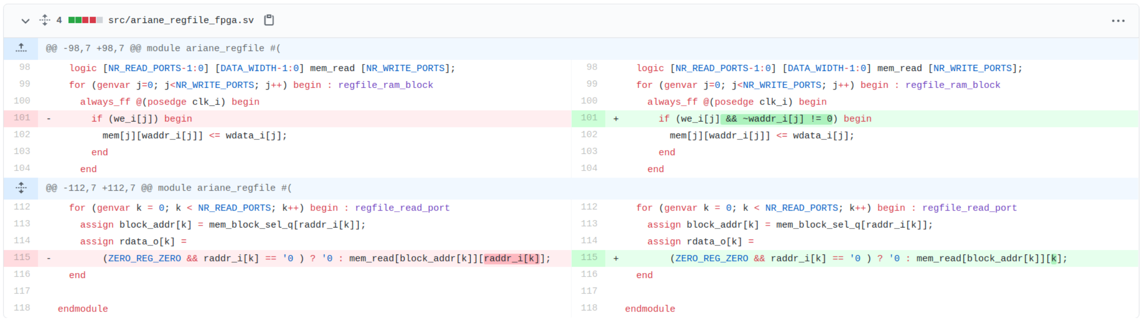
One solution for this problem is to execute on the first half of the clock the two write operations on the same memory and, on the second half of the clock period, execute the reads. This could be achieved by using a TDP memory with double the clock frequency, having a switching logic to modulate both operations. This solution wasn't explored further, as some meta-stability problems could arise, but theoretically it could be achieved, as the maximum operation frequency of the memory is way higher than the current processor operating frequency.

Another solution to this problem is to duplicate the register file using two TDP memories to operate on four addresses on the same time. With that implementation, a mapping of the physical location of the information needs to be done and updated at each write operation. This ensures that the read operation can be executed correctly. The cost of such mapping on the current implementation is one bit per address, that means, 32 bits. A important point to be made is that write operations are independent of physical location, as the mapping will always point to the correct memory block after a write operation. That said, after having both read operations allocated in function of the current mapping of the addresses, the write operations can be allocated to the available ports, no matter their location. Of course that theoretically a read and a write operation could happen on the same address, but considering the forwarding block on the processor, such problem is automatically corrected. Table 1 shows the results of such implementation.

2.1.2 Synthesizing with LUTRAM

GitHub branch: `regfile_fpga`.

We do not claim authorship of the code for the LUTRAM register file. The original code was discovered in the commit history of the original repository. The solution consists in replacing flip flops for distributed RAM in the FPGA matrix. However, the commit was not functional (and hence reverted). The problem was due to a badly written memory indexation. We provide a fix for the bug in the code. The fix is shown in Figure 2.1.



```
4 src/ariane_regfile_fpga.sv
```

```
@@ -98,7 +98,7 @@ module ariane_regfile #(
    logic [NR_READ_PORTS-1:0] [DATA_WIDTH-1:0] mem_read [NR_WRITE_PORTS];
    for (genvar j=0; j<NR_WRITE_PORTS; j++) begin : regfile_ram_block
      always_ff @(posedge clk_i) begin
-       if (we_i[j]) begin
          mem[j][waddr_i[j]] <= wdata_i[j];
        end
      end
@@ -112,7 +112,7 @@ module ariane_regfile #(
    for (genvar k = 0; k < NR_READ_PORTS; k++) begin : regfile_read_port
      assign block_addr[k] = mem_block_sel_q[raddr_i[k]];
      assign rdata_o[k] =
-         (ZERO_REG_ZERO && raddr_i[k] == '0 ) ? '0 : mem_read[block_addr[k]][raddr_i[k]];
    end

  endmodule
```

```
    logic [NR_READ_PORTS-1:0] [DATA_WIDTH-1:0] mem_read [NR_WRITE_PORTS];
    for (genvar j=0; j<NR_WRITE_PORTS; j++) begin : regfile_ram_block
      always_ff @(posedge clk_i) begin
+       if (we_i[j] && ~waddr_i[j] != 0) begin
          mem[j][waddr_i[j]] <= wdata_i[j];
        end
      end

    for (genvar k = 0; k < NR_READ_PORTS; k++) begin : regfile_read_port
      assign block_addr[k] = mem_block_sel_q[raddr_i[k]];
      assign rdata_o[k] =
+         (ZERO_REG_ZERO && raddr_i[k] == '0 ) ? '0 : mem_read[block_addr[k]][k];
    end

  endmodule
```

Figure 2.1 – Bug fix for LUTRAM register file

Using such a register file implementation provides gain in terms of the number of LUTs and Flip Flops used in the system. Table 1 shows the results for that branch, alongside with the results of the BRAM register file.

	Max Frequency	Total LUTs	Logic LUTs	Total Flip Flops	LUTRAMs	Block RAM (36 kB)
Original Code	52.09 MHz	14631	14631	9286	0	36
LUTRAM Register File (regfile_fpga)	50.64 MHz (-2.78%)	13730 (-6.16%)	13634 (-6.81%)	8326 (-10.34%)	96	36
BRAM Register File (regfile_tdp_bram)	52.27MHz (+0.34%)	13745 (-6.05%)	13745 (-6.05%)	8460 (-8.89%)	0	38

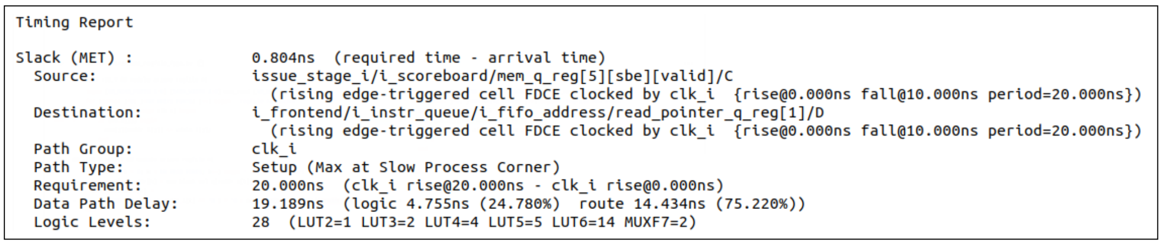
Table 1 – Maximum frequency and utilization results for different register file implementations

2.2 Adding flip flops to optimize the worst path delay

2.2.1 Signal flush_i on module serdiv

Branch on GitHub: `srdiv_flush_register`

As shown in Figure 2.2, by letting the system clock period be equal to 20ns, the worst slack for setup timings is 0.804ns, which allows the core to run at a theoretical frequency of $f = \frac{1}{(20-0.804)ns} = 52,094MHz$. Note that in order to optimize the max frequency in the core, we are interested in setup rather than hold timings.



```
Timing Report

Slack (MET) :              0.804ns  (required time - arrival time)
  Source:                  issue_stage_i/i_scoreboard/mem_q_reg[5][sbe][valid]/C
                             (rising edge-triggered cell FDCE clocked by clk_i  {rise@0.000ns fall@10.000ns period=20.000ns})
  Destination:             i_frontend/i_instr_queue/i_fifo_address/read_pointer_q_reg[1]/D
                             (rising edge-triggered cell FDCE clocked by clk_i  {rise@0.000ns fall@10.000ns period=20.000ns})
  Path Group:              clk_i
  Path Type:               Setup (Max at Slow Process Corner)
  Requirement:             20.000ns  (clk_i rise@20.000ns - clk_i rise@0.000ns)
  Data Path Delay:         19.189ns  (logic 4.755ns (24.780%)  route 14.434ns (75.220%))
  Logic Levels:            28  (LUT2=1 LUT3=2 LUT4=4 LUT5=5 LUT6=14 MUXF7=2)
```

Figure 2.2 – Worst slack in original code

The worst path shown in the figure is identified as a flush control flow. More specifically, the FIFO `i_fifo_address`, instantiated in the module `i_instr_queue`, in the processor front-end, uses the `flush_i` input signal in a conditional clause. The computation of this signal comes from the scoreboard, and goes through exactly 28 logic levels until it finally reaches the front-end again.

By tracking the flush signal through the path, we choose to insert a flip flop on the exact midpoint of the path, which happens to be the module `serdiv`. Listing 1 shows the flip flop insertion (with misplaced line numbers).

Listing 1 – Insertion of a flip flop in module serdiv

```
1  logic flush_q;
2  always_ff @(posedge clk_i) begin : flush_register
3    flush_q <= flush_i;
4  end
```

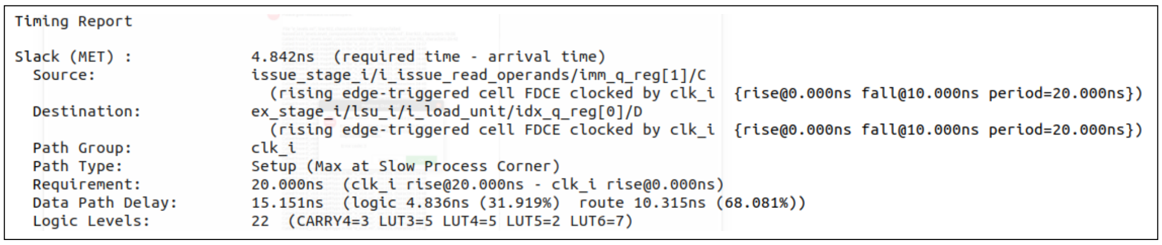
Although the change adds one cycle of latency for that instruction flow specifically, it does not impact on coremark. The slack for 20ns clock period goes up to 4.82ns, which allows the maximum theoretical frequency to be significantly higher.

Table 2 shows the results for that change.

2.2.2 Signal page_offset_matches_i on module load_unit

Branch on GitHub: `loadunit_pageoffset_flop`

We consider the new worst path obtained after applying the branch described on section 2.2.1. It can be seen in Figure 2.3. Following the same methodology, we insert a flip flop in the middle of it. We choose to insert the flip flop in the module `load_unit`, on the input signal `page_offset_matches_i`. Surprisingly, this not only raised the frequency by a small amount but also lead to a increased coremark score. Results can be seen in Table 2.



```
Timing Report

Slack (MET) :            4.842ns  (required time - arrival time)
  Source:                issue_stage_i/i_issue_read_operands/imm_q_reg[1]/C
                           (rising edge-triggered cell FDCE clocked by clk_i  {rise@0.000ns fall@10.000ns period=20.000ns})
  Destination:           ex_stage_i/lsu_i/i_load_unit/idx_q_reg[0]/D
                           (rising edge-triggered cell FDCE clocked by clk_i  {rise@0.000ns fall@10.000ns period=20.000ns})
  Path Group:            clk_i
  Path Type:             Setup (Max at Slow Process Corner)
  Requirement:           20.000ns  (clk_i rise@20.000ns - clk_i rise@0.000ns)
  Data Path Delay:       15.151ns  (logic 4.836ns (31.919%)  route 10.315ns (68.081%))
  Logic Levels:          22  (CARRY4=3 LUT3=5 LUT4=5 LUT5=2 LUT6=7)
```

Figure 2.3 – Worst slack after applying change described in section 2.2.1

	Max Frequency	CoreMark	Total LUTs	Logic LUTs	Total Flip Flops	LUTRAMs	Block RAM (36 kB)
Original Code	52.09 MHz	112.2083	14631	14631	9286	0	36
flip flop insertion on serdiv	65.97 MHz (+26.64%)	112.2083	14631	14631	9287 (+1)	0	36
flip flop insertion on load_unit and serdiv	66.24 MHz (+27.16%)	112.5486	14632 (+1)	14632 (+1)	9288 (+2)	0	38

Table 2 – Maximum frequency and utilization results after flip flop insertions

2.3 Rewriting combinational logic inside the scoreboard

Micro-optimizations can be made on the code to improve the synthesis results, for example replacing ternary operators with bit operators. It was found that by doing this, an (small) improvement on the overall slack of the processor could be achieved. Table 3 shows the (small) improvement on the frequency of the processor. Unfortunately, this comes at the price of readability specially on the multiple bits case. This can be seen on the code bellow, where a ternary assignment is replaced by multiple and operators.

Listing 2 – Rewriting ternary assignment with bitwise and operator

```
// Original code
assign commit_pointer_n[k] = (flush_i) ? '0 : commit_pointer_n
    [0] + unsigned'(k);

// Micro-optimization
wire [BITS_ENTRIES-1:0] commit_pointer_n_tmp =
    commit_pointer_0_tmp + unsigned'(k);

for(genvar ii=0; ii < BITS_ENTRIES; ii++) begin
   assign commit_pointer_n[k][ii] = (~flush_i) &
       commit_pointer_n_tmp[ii];
end
```

	Max Frequency	**Total LUTs**	**Logic LUTs**	**Total Flip Flops**	**LUTRAMs**	**Block RAM (36 kB)**
Original Code	52.09 MHz	14631	14631	9286	0	36
Micro-optimization	52.32 MHz (0.5%)	14648 (+0.1%)	14648 (+0.1%)	9285 (0%)	0	36

Table 3 – Micro-optimization done on the scoreboard by replacing the ternary assignment to a logic and

2.4 Increasing the size of critical blocks

By analyzing the performance counters on coremark execution, we conclude that performance degradation comes mainly from **branch mispredictions** and **instruction cache misses**. Therefore, one way to achieve better IPC is to augment the size of critical blocks/structures that impact on the performance the most. For instance, we could augment the number of entries of both BTB and BHT, the size of the instruction cache, its associativity or its line size. Table 4 shows the impact on performance of augmenting the number of BHT, BTB and RAS entries by a 4 factor, and doubling the size of the instruction cache line, doubling the overall size of the instruction cache and doubling its associativity.

	CoreMark	**Branch mispredicts**	**L1 Instruction Cache Misses**
Original Code	112.2082	33119	3989
x2 L1 I$ Cache Line x2 L1 I$ Size x2 L1 I$ Associativity x4 BHT Entries x4 BTB Entries x4 RAS Entries	112.6783	31816 (-3.93%)	1690 (-57.63%)

Table 4 – Coremark execution : branch mispredicts and instruction cache misses

Clearly, the drawback is a higher usage of LUTs and flip flops.

As stated in section 1, memory synthesis is one of the main bottlenecks when it comes to ASIC vs FPGA designs. Therefore, the strategy of redesigning modules to use dedicated FPGA memory (BRAM or LUTRAM) in order to economize on the number of LUTs and flip flops gives room for size augmentation of performance critical blocks.

We did not include such size modifications on our final results, since we consider size gain as an optimization factor that can be later used in different ways to increase overall performance.

3 Results

The results were assembled in two different branches: `final_bram` and `final_lutram`. While the former uses the BRAM version of the register file, the latter uses the LUTRAM version.

We consider our final results to be the ones given `final_lutram` by the branch on. It is a merge of the following individual optimisations:

`regfile_fpga`;

`scoreboard_mod`;

`srdiv_flush_register`;

`loadunit_pageoffset_flop`

The results can be seen in Table 5.

		Original Code	**Our Results (final_lutram)**	**Percentage**
Timing	Worst Slack (20ns)	0.804ns	4.869ns	
	Max Frequency	52.09 MHz	66.01 MHz	**+ 22.72%**
CoreMark	CoreMark Score	112.21	112.55	
	Simulation Frequency	50 MHz	50 MHz	
	CoreMark / MHz	2.244	2.251	**+ 0.31 %**
Utilization	Total LUTs	14631	13748	**- 6.03 %**
	Logic LUTs	14631	13652	**- 6.69 %**
	LUTRAMs	0	96	
	FFs	9286	8327	**- 10.33%**
	BRAM	36	36	
	SRLs	0	0	
	DSP	4	4	
Style	Number of changes		5 files 165 added lines 9 deleted lines	

Table 5 – Final Results

Although it is **not considered to be** our final result, we also present the final results considereing the BRAM register file implementation. The branch `final_bram` is a merge of the following branches:

`regfile_tdp_bram`;

`scoreboard_mod`;

`srdiv_flush_register`;

`loadunit_pageoffset_flop`

Using the BRAM register file with the other branches made us loose the frequency gains, while we could keep those gains by using the LUTRAM register file. That is the reason why we chose the LUTRAM register file as our final result. The condensed results using the BRAM register file can be seen in Table 6.

	Timing		**Coremark**	**Utilization**		
	Worst Slack (20ns)	**Max Frequency**	**Score**	**Total LUTS**	**Logic LUTs**	**FFs**
Original Code	0.804ns	52.09 MHz	112.208259	14631	14631	9286
final_bram	4.27ns	63.57 MHz (+22.04%)	112.548649 (+0.3%)	13764 (-5.93%)	13764 (-5.93%)	8463 (-8.86%)

Table 6 – Final results - BRAM register file

4 Other things we tried to do

We have also tried some other modifications, which due to diverse constraints, such as limited personnel and time, we could not finish.

4.1 Change memory implementation in other places

There are other structures that consume a considerable amount of LUT and FF resources, such as the BTB, the BHT and the Scoreboard. We started a implementation of the BTB to be synthesized with LUTRAM, using the modules provided in `src/fpga-support/rtl`. Finishing this optimization would result in significant resource gains, and thus give room for size improvement in other critical parts.

4.2 Use Xilinx Unimacro Library

Branch on GitHub: `fifo_mod`

The control signal path between the FIFOs in the frontend and the backend builds for the worst timings in the core. By replacing the module `fifo_v3` for a off-the-shelf FIFO implementation, much of the logic regarding FIFO management is optimized, thus resulting in better timings. Xilinx provides a FIFO implementation with block RAM in its `unimacro` library, in asynchronous and synchronous versions. We explore the use of that module. Due to the fact that the module `fifo_v3` is instantiated in different parts of the code with variable data types, we did not manage to make the solution work, since we would need to split the input data into multiple FIFOs. However, this remains as a interesting optimization to be further explored.

Tough simulation did not work, we reached a maximum frequency of 83MHz while synthesizing the code in the indicated branch.

5 Conclusion

First of all, the competition served its learning purpose, as we, as a team, were able to dig into the code and look for possible improvements. The knowledge we acquired touched the areas of engineering (problem solving), team working, micro-architecture, simulation, synthesis and others.

As for our results, we successfully met the criteria, as we were able to augment the frequency, the overall coremark and reduce the number of used resources. However, we acknowledge that the improvements could be much larger. We are left with the feeling that by the end, when we finally deployed some actual optimizations, not only we had a reduced team, but not much time left was left. In other words, there was a slow learning curve in the beginning regarding which direction to look at.

References

[1] David Sheldon and Frank Vahid. Don't forget memories: A case study redesigning a pattern counting asic circuit for fpgas. In *Proceedings of the 6th IEEE/ACM/IFIP international conference on Hardware/Software codesign and system synthesis*, pages 155–160, 2008.

[2] F. Zaruba and L. Benini. The cost of application-class processing: Energy and performance analysis of a linux-ready 1.7-ghz 64-bit risc-v core in 22-nm fdsoi technology. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 27(11):2629–2640, Nov 2019.