Cambridge University Engineering Department
Engineering Tripos Part IIA
PROJECTS: Interim and Final Report Coversheet

IIA Projects

TO BE COMPLETED BY THE STUDENT(S)

Project:	GB3 – RISC-V Processor		
Title of report:	Individual Report		
Name(s): (capitals)		crsID(s):	College(s):
WILL HEWES		wh365	Pembroke

Declaration for: Final Report

I confirm that, except where indicated, the work contained in this report is my own original work.

Instructions to markers of Part IIA project reports:

Grading scheme

Grade	A*	A	B	C	D	E
Standard	Excellent	Very Good	Good	Acceptable	Minimum acceptable for Honours	Below Honours

Grade the reports by ticking the appropriate guideline assessment box below, and provide feedback against as many of the criteria as are applicable (or add your own). Feedback is particularly important for work graded C-E. Students should be aware that different projects and reports will require different characteristics.

Penalties for lateness: Interim Reports: 3 marks per weekday; Final Reports: 0 marks awarded – late reports not accepted.

Guideline assessment (tick one box)

A*/A	**A/B**	**B/C**	**C/D**	**D/E**

Marker:		Date:	

Delete (1) or (2) as appropriate (for marking in hard copy – different arrangements apply for feedback on Moodle):

(1) Feedback from the marker is provided on the report itself.

(2) Feedback from the marker is provided on second page of cover sheet.

	Typical Criteria	Feedback comments
Project Skills, Initiative, Originality	Appreciation of problem, and development of ideas	
	Competence in planning and record-keeping	
	Practical skill, theoretical work, programming	
	Evidence of originality, innovation, wider reading (with full referencing), or additional research	
	Initiative, and level of supervision required	
Report	Overall planning and layout, within set page limit	
	Clarity of introductory overview and conclusions	
	Logical account of work, clarity in discussion of main issues	
	Technical understanding, competence and accuracy	
	Quality of language, readability, full referencing of papers and other sources	
	Clarity of figures, graphs and tables, with captions and full referencing in text	

ENGINEERING TRIPOS PART IIA

GB3 - Risc-V Processor

Final Report

Group 4 - Resource Usage

Will Hewes - wh365
Pembroke College

Contents

1 Introduction

This project involves the collaborative design and implementation of a RISC-V processor on an iCE40 FPGA device. The overarching aim is to optimise the processor design with respect to its performance, power dissipation, and resource usage, while balancing trade-offs between these three aspects. Each team member is responsible for focusing on one of the three aspects, and my reports focus on the resource usage of the design. This report is the final report for this project.

In the first report we evaluated three programs as a baseline against which our improvements could be compared. The first of these three programs was `hardwareblink`, a program which relied purely on combinational and sequential logic to blink an LED on the FPGA, bypassing the processor core entirely. As a result it had very low resource utilisation across the board. The second program evaluated was `softwareblink`, which performed the same task but instead implemented through software, utilising the sail-core processor. As a result, the design used a much larger number of logic cells, and also introduced the usage of a few other primitive cells such as block RAMs and global buffers. The final program was `bubblesort`, which would perform a bubblesort algorithm and blink the LED in a pattern corresponding to its sorting process. Due to the increased complexity of the program, the logic cell utilisation was greater than for `softwareblink`, but crucially the other primitives such as block RAMs and DSPs did not increase between programs, as their usage is fixed by how they are instantiated in the Verilog design, rather than varying with program complexity. The final test would be to process a new, unseen program under our implementation and evaluate it for its resource usage, performance and power dissipation. As such we chose to base our testing primarily on the more complex `bubblesort`, allowing us to push the processor further, offering a more representative view of our design. It should be noted the other programs were still tested to ensure functionality and check progress, they were just less emblematic of our implementations capabilities.

The baseline implementation on the FPGA showed several inefficiencies in its core design and made poor use of some of the resources available on the board. Before outlining the changes made, it's useful to first explain what these resources are and how they influence the processor's behaviour.

Logic cells, or LCs, are the basic building blocks used to implement logic on the FPGA. They include components like look-up tables and flip-flops, and are used for almost everything, including arithmetic, control signals, and registers. Because almost everything in the design depends on logic cells in some way, reducing how many are used was the main goal throughout the project. The changes that directly improved LC usage primarily came from cleaning up registers, wires, and other redundant logic where possible. One notable example of this was reducing a multiplexing wire signal from 32 bits down to 11 bits. These examples will be discussed in more detail in the body of the report.

Other primitives available on the board are similarly powerful, though they have a key difference - they are fixed for a given implementation on the microprocessor. This means that whereas LCs will increase and potentially become limiting for a particularly complex program, other primitives such as block RAMs, DSP registers, and buffers will not dynamically increase outside of the capacity of the board if designed appropriately.

DSP blocks are specialised units designed for fast arithmetic. Instead of building adders and multipliers out of regular logic, the FPGA provides a few dedicated DSP blocks that are faster and use less space. The baseline design didn't use any of these, relying entirely on logic cells even for basic addition. One of the most significant alterations to the base implementation was forcing the *adder.v* module to make use of these DSP blocks for addition and subtraction, freeing up LUTs to be used in more sophisticated logic. Implementing this change reduced the LC usage by 94, utilising 3 of the 8 DSP blocks available, leaving room for further expansion.

Block RAMs are small dedicated memory blocks built into the FPGA. They're useful when a design needs to store more data than would be practical with flip-flops alone. In the baseline implementation, they were used by certain parts of the processor such as the register file and CSR logic. By removing CSR logic from the design, the number of block RAMs used was reduced from 20 to 12 of the 30 available, meaning that more memory functions could be moved to the RAM in order to preserve logic cells. Although time constraints prevented further exploitation of this freed-up memory, several changes were planned to take advantage of the available space and further reduce logic usage. These included simplifying large logic blocks using Verilog `generate` statements to reduce code duplication, sharing comparator logic in the forwarding unit to cut down on redundant circuitry, cleaning up control signal wiring around the register file, and offloading additional arithmetic operations to the FPGA's DSP blocks. These modifications, if implemented, would help reduce pressure on logic cells and potentially improve both timing and power efficiency.

In addition to these changes that were successfully implemented, there are many that could not be pursued within the time constraints of the project. These include more aggressive pipelining of the execution stages, a partial redesign of the register file interface, and further refinement of control logic to reduce signal fanout. While these ideas remain untested, they represent promising avenues for future exploration, particularly in balancing resource usage with performance and scalability. The following sections provide a detailed breakdown of the changes that were implemented, their technical justification, and the impact each had on the final design.

2 Design Strategy

As I was initially unfamiliar with processor and FPGA design at this level, the early stages of the project were spent working through the course material and exploring online resources to build a clearer understanding of where changes could be most effectively targeted. This initial research was followed by a series of small changes to the Verilog code, helping build familiarity with the processor's structure and behaviour. In hindsight, these early edits were made without a proper testing procedure in place, but they laid the groundwork for a more focused and informed optimisation approach that developed as the project progressed. This unstructured approach naturally led to a few early mistakes, such as unintentionally breaking functionality or introducing hard to trace bugs. These mistakes will be discussed more in Section 4. These issues highlighted the need for a more disciplined workflow, which later included stricter version control, more careful integration with other modules, and simulations to verify that signals behaved as expected after changes.

A significant part of the optimisation process involved carefully examining the Verilog source files to understand the structure and interaction of each component across multiple files. This required tracing signal flow through the datapath and control logic, identifying where logic or memory resources were being used inefficiently, and confirming whether particular subsystems - such as CSRs or multiplexer paths - were functionally active during execution. Pattern-matching tools like `grep` were used extensively to locate signal definitions and check bit widths, while simple simulation testbenches were created to verify specific behaviours, most notably during the laboured construction of the DSP blocks. These methods helped build a reliable picture of the design's internal behaviour and revealed opportunities to safely eliminate or simplify hardware blocks without compromising core functionality.

As a team, we collaborated using Git, setting up individual branches for each member to work on their specific optimisation focus. This allowed us to implement and test changes independently, while maintaining a clean and stable main branch. Once inefficiencies were identified and addressed within a branch, the updates could be merged cohesively with others. This workflow not only supported parallel development but also helped manage merge conflicts more effectively as the project progressed.

Unfortunately, this is another aspect where planning would have made a significant improvement to our working procedure. As there was only one FPGA, testing our designs on the board was not always possible, so our individual changes would be applied and though they would successfully synthesise and be routed, often the functionality would be broken and we would be unable to tell until the code could be tested on the FPGA when we would all meet again. This meant that much of our time was spent working on changes that would need to be extensively debugged, and unfortunately many of these initial changes would in the end remain unimplemented due to time constraints, most notably the pipelining changes within the *adder.v* and *alu.v* modules.

3 Design Description

The changes made during this project were, for the most part, minor and locally scoped optimisations rather than sweeping architectural overhauls. The aim throughout was to reduce resource usage on the FPGA - specifically the logic cell count - without compromising the functional correctness of the processor. While logic cells were the primary focus, some changes also impacted the usage of other board primitives such as block RAMs and DSP blocks.

Each optimisation was applied as discussed in earlier sections, incrementally applied and tracked using Git, with synthesis reports logged at every stage to assess their effect. The following subsections outline each of these changes, along with their motivation, implementation, and observed impact. Where relevant, small pieces of code are shown in the main body of the report, while longer or supporting code is included in the appendix.

3.1 Clock Gating

One of the earliest changes involved modifying the program counter (PC) logic to reduce unnecessary switching activity during pipeline stalls. In the baseline implementation, the PC would increment on every clock cycle regardless of whether the pipeline was stalled, resulting in wasted transitions and toggling of downstream logic even when no useful computation was occurring.

To address this, support for gated updates to the program counter was introduced. This involved modifying the *program_counter.v* module to accept a new `write_enable` signal, allowing updates to be conditionally applied only when pipeline progression was required. The *cpu.v* module was updated to assert this signal whenever the `stall` flag was low, ensuring that the PC would remain stable during pipeline stalls. The signal was then routed through *top.v* to maintain module connectivity.

This change primarily aimed to reduce dynamic power dissipation by avoiding unnecessary switching in the

PC logic. While the overall resource saving was modest - freeing approximately 9 logic cells - it represents an important step in selectively controlling register updates. It also clarified the flow of stall logic through the processor and established a cleaner separation between control and datapath behaviour.

One trade-off was a slight increase in global buffer (GB) usage, rising from 5 to 6 out of the 8 available on the iCE40 device. Global buffers are specialised routing resources used to distribute high-fanout signals - such as clocks or enables - efficiently across the FPGA fabric. Their usage ensures consistent signal timing over long distances, especially when a control signal must drive registers in multiple modules. The increase was likely due to the new `pc_write_enable` signal being propagated through several layers of control and datapath logic, requiring global routing resources to maintain timing closure. This is not a key resource in the same way that LUTs and flip-flops are, and so a minor increase in GB usage is not considered critical. Provided the design remains within the hardware limit of 8, such changes are acceptable if they simplify control signalling or improve timing.

The impact of this modification is summarised in Table 3 in the Appendix.

3.2 Signal Width Reduction

Another optimisation involved reducing the width of a control signal used in the execution-to-memory pipeline stage. This signal, `ex_cont_mux_out`, is used to carry control values from the decode stage into the execution stage, particularly for managing behaviours such as branching, memory operations, and ALU control. In the baseline implementation, this signal was declared as 32 bits wide, despite the fact that only the lower 11 bits were ever referenced or extracted in downstream modules. This resulted in an oversized bus that incurred extra logic and routing costs without delivering any functional benefit.

To identify the mismatch, `grep` was used to trace the usage of `cont_mux_out` throughout the Verilog files. This revealed that only specific slices of the signal were ever used, confirming that the top 21 bits could be safely discarded. With this knowledge, a dedicated 11-bit multiplexer module, `mux2to1_11bit`, was written and substituted alongside the generic 32-bit `mux2to1`. The change required updating the module instantiation and associated wiring, but left the logic semantics unchanged. This initial implementation is shown in Program 7 in the Appendix for completeness.

After implementing this successful change, reducing logic cell usage but maintaining functionality of the program, this was expanded upon to make this approach more flexible. The `mux2to1` assignment was expanded to support a parameterised width, allowing the same module to be reused in other parts of the design without hardcoding the bit width. [1] This is shown in Program 1 below.

```
module mux2to1_param #(parameter WIDTH = 32)
( // Example instantiation: mux2to1_param #(.WIDTH(32))
    input  [WIDTH-1:0] input0,
    input  [WIDTH-1:0] input1,
    input              select,
    output [WIDTH-1:0] out
);
    assign out = (select) ? input1 : input0;
endmodule
```

Program 1: Generic width multiplexer

This allows bit widths to be specified for each instantiation of the multiplexing operation, reducing redundancy in the assignment of wires and registers. This improvement reduced logic cell usage by 25 LUTs, as shown in Table 4 in the Appendix.

It should be noted that some widths were kept at 32 bits, despite possessing redundancy in their width. These were left unchanged as, although reducing their width would result in slightly fewer LUTs being used, leaving their widths at 32 bits with the generic parameter multiplexing ended up reducing the delay from around 76ns to 69ns, producing an unnaccounted for reduction in delay of 10%. In the future, this consequence would have to be probed further to determine the root cause, but due to time constraints this was not possible. There may be a more optimal configuration that could reduce resource usage and improve timing further, but this was not fully explored.

3.3 DSP Substitution

Another important optimisation involved offloading arithmetic logic from general-purpose logic cells to dedicated DSP blocks on the FPGA. In the baseline implementation, the adder logic used for core operations such as basic arithmetic and branch comparisons was fully implemented in LUT-based logic, consuming a significant number of logic cells and contributing to routing congestion.

To address this, the adder was restructured to use the `SB_MAC16` primitive provided by the iCE40 FPGA, which includes dedicated adder functionality alongside additional capabilities that were not utilised in this implementation.

In order to shift arithmetic operations over to the DSP, the `SB_MAC16` has to be explicitly declared, otherwise the synthesis tools will just use general purpose LUTs instead. This was implemented in the *adder.v* module, where the 32-bit operands were split into upper and lower 16-bit segments and mapped to the DSP's internal A, B, C, and D inputs. Parameterisation via `defparam` was used to configure the carry chain and data routing between the top and bottom halves of the DSP. This is displayed in Program 8 in the Appendix.

This module was then instantiated within the ALU, allowing RISC-V instructions that required add operations to route through the DSP instead of a combinational adder. In order to implement addition, initially a wire was created that could hold the 2s complement of a wire, as shown below in Program 2.

```
wire [31:0] B_neg = ~B + 1;
```

Program 2: 2s complement

However, this unnecessarily increased complexity, and so instead the control signal `is_sub` was added to dynamically toggle between addition and subtraction by setting the `ADDSUBTOP` and `ADDSUBBOT` inputs within the `SB_MAC16`.

This is then instantiated in module *alu.v* as shown below in Program 3 below.

```
/*
 *       ADD
 */
`kSAIL_MICROARCHITECTURE_ALUCTL_3to0_ADD:       begin
        do_sub = 1'b0;
        ALUOut = add_result;
end
/*
 *       SUBTRACT
 */
`kSAIL_MICROARCHITECTURE_ALUCTL_3to0_SUB:       begin
        do_sub = 1'b1;
        ALUOut = add_result;
end
```

Program 3: is_sub implementation within alu.v

Following this change, logic cell usage was reduced by 94 LUTs, at the cost of 3 out of 8 available DSP blocks. As mentioned in Section 1, this increase in the usage of DSPs is not a negative, as they are fixed hardware components that would otherwise remain unused. Shifting the simple arithmetic operations from far more malleable LUT and flip-flops over to the specialised DSPs not only expands the capacity for software complexity on the board but also improves the performance, as DSPs are optimised for fast arithmetic calculations.

A simplified diagram illustrating the data path through the DSP adder is provided in Figure 1 in the Appendix.

3.4 CSR

The most impactful change made to the processor was the complete removal of the Control and Status Register (CSR) logic. CSRs are a core but optional component of the RISC-V architecture specification, primarily used to support privileged operations such as trap handling, performance counters, environment configuration, and interrupt control. In a full system, CSR instructions are essential for switching between user and machine modes, managing exceptions, and interacting with the operating system. [2]

However, for the purposes of this project, none of the benchmark programs utilised any CSR-related functionality. The processor ran entirely in *machine mode*, the highest privilege level defined in the RISC-V specification. In this simplified configuration, there was no support for exception handling, privilege escalation, or multiple execution contexts. As none of these CSR features - such as interrupt detection, configuration control, or capability reporting - were used by the benchmark programs, the CSR subsystem was never accessed during execution and could be safely removed without impacting functionality. Despite this, the baseline implementation included a fully integrated CSR register file backed by block RAM, associated decode logic in the control unit, and forwarding and hazard-handling support for CSR instructions.

This was identified as redundant after checking that CSR logic was never triggered during program execution. Initial attempts at removal simply set all the relevant function codes and carrier signals to zero, introducing dummy wires to ensure the CSR logic was removed without making largescale changes. This is demonstrated in Program 4 below, displaying one instance in which the CSR logic was initially nullified.

```
wire                    CSRR_signal = 1'b0;
wire                    CSRRI_signal = 1'b0;
```

Program 4: Short example of nullifying CSR logic

Eventually, all references to CSR were removed using `grep` to locate each instance across the codebase. Because CSR logic was spread throughout the design - including the defines files, ALU, control unit, forwarding logic, and CPU - this process required several passes to fully clean up. The dedicated *csr.v* module was also deleted, completing the removal and yielding a meaningful reduction in resource usage.

Removing the CSR logic significantly simplified the processor architecture and yielded the largest resource reductions in the entire project. The CSR register file alone occupied 8 of the 30 available block RAMs on the iCE40 device, and its associated decode, hazard handling, and forwarding logic introduced considerable complexity into the control and execution paths. By eliminating this subsystem entirely the design achieved a reduction of 251 logic cells and 8 block RAMs, as reflected in Tables 1 and 6 in the Appendix. These gains not only freed critical resources, but also opened the door for relocating other memory elements into block RAM in future, improving the flexibility of the implementation. This change exemplifies the benefit of tailoring FPGA designs to their target workloads: by removing unused ISA features like CSR support substantial savings were achieved without compromising correctness or functionality. This change also introduced a minor increase in global buffer usage, as removing the CSR subsystem altered control signal routing and clock enable logic.

Looking ahead, the resources freed by CSR removal created opportunities for further optimisation. In particular, additional arithmetic operations could be shifted onto the remaining DSP blocks, reducing pressure on LUT-based logic. Similarly, with block RAM usage significantly reduced, some logic structures or lookup tables could be restructured to use memory elements instead, freeing up logic cells and potentially improving timing and power efficiency.

4 Problems Encountered

A number of challenges were encountered throughout the optimisation process, ranging from debugging difficulties to integration issues. This section outlines the most significant obstacles, the strategies used to address them, and steps that could be taken to mitigate similar issues in future projects.

Limited Access to Hardware

Hardware testing was an essential part of the validation process, but it came with several practical constraints. Only one FPGA board was available to the group, which naturally restricted testing opportunities. This limitation often caused delays between implementing a change and receiving confirmation of success or failure. While synthesis and place-and-route tools can flag syntax or timing issues, they could not guarantee correct functionality. As a result, functionality sometimes appeared intact post-synthesis, only to fail on deployment at a later testing point.

In several early cases, multiple untested commits were made sequentially before the hardware could be tested. This led to the uncovering of errors in several commits after they were introduced, requiring backtracking through Git to isolate the root cause - this was an inefficient and meandering process.

A more sensible approach, which we began implementing towards the end, was to separate every single untested change into a distinct branch, using version control to its fullest extent. When we had access to the hardware, we could implement our changes as before, testing each change as it was added, but when we were unable to check our changes we would instead build from a known working point and branch off from there, ensuring minimal risk of cascading failures.

Unfortunately, this issue was especially prevalent with regard to the performance-related modifications made at the start of the project. Many of these involved deep changes to pipelining and hazard handling logic, which were made in bulk and committed before hardware testing was properly completed. When the LED blink output subsequently failed, it became difficult to isolate which part of the update had caused the lapse in functionality.

Despite significant effort spent debugging and attempting to reintegrate these changes, the complexity and lack of intermediate validation meant they could not be reliably restored within the project timeline. As a result, several potential performance improvements were ultimately abandoned to preserve stability in the final implementation, notably much of the planned changes to pipelining.

Lack of Formal Simulation

At the beginning of the project, little to no formal simulation was performed. Instead, hardware testing was used as the main method of verification, which meant even trivial bugs required full synthesis and flashing to detect. This proved inefficient, especially when changes affected multiple interconnected modules.

A turning point came during the implementation of DSP block usage in the adder module. This was one of the most technically challenging modifications. As mentioned in Section 3, previously the *adder.v* file had been very simple, implementing addition as shown below in Program 5, relying on a combinational adder.

```
module adder(input1, input2, out);
        input [31:0]    input1;
        input [31:0]    input2;
        output [31:0]   out;

        assign          out = input1 + input2;
```

Program 5: Initial adder.v module

In order to replace this, all of the parameters for the DSP had to be declared, either explicitly or by reverting to the implicit values of the primitive. The information for this implementation largely came from the DSP function usage guide [3], which detailed the parameters that could be set and the structure of the block. Figure 1 in the Appendix displays a diagram from the guide illustrating the structure of the DSP. The full *adder.v* module is shown in Program 8 in the Appendix.

Substituting in the SB_MAC16 primitive required careful parameter configuration and correct wiring of carry chains between the top and bottom halves of the DSP. Unfortunately, official documentation lacked clear examples for 32-bit adders, leading to misconfigured carry propagation when first implementing this change.

Although the updated adder passed hardware tests with simple programs like `softwareblink`, it silently failed on `bubblesort`, producing incorrect output patterns partway through execution. Because the bug was subtle and did not manifest clearly, it evaded detection until several unrelated changes had been committed on top. This resulted in significant debugging effort, including rolling back and isolating the root cause.

In the process of debugging, module-level simulation was adopted. Testbenches were created for units such as the *alu.v* and *adder.v* modules, which enabled more reliable verification. Although coverage remained limited, this approach caught issues earlier in development and dramatically reduced the turnaround time for identifying functional errors.

This experience highlighted the value of incorporating simulation early, especially when working with lower-level FPGA primitives where subtle misconfigurations can lead to difficult to trace bugs.

Integration and Merging Difficulties

Working in parallel across multiple branches introduced challenges when merging changes. Key files such as *cpu.v* and *control_unit.v* were edited by multiple team members, frequently leading to merge conflicts or subtle behavioural bugs. The difficulty was exacerbated by the tight coupling between datapath and control logic, which meant that even small changes could have unexpected ripple effects.

To mitigate this, a more disciplined Git workflow was adopted. Changes were tested individually and only merged into shared branches after validation. This reduced errors and provided a stable baseline for collaborative development.

5 Test Procedure

Primary Workflow

My workflow initially consisted of running the various `Makefile` commands to initiate yosys and nextpnr, synthesising our implementation and performing a place-and-route operation. I saved the reports from these operations to a .txt file, allowing me to perform *git diff* on the text file from any of my commits to easily see the impact that each change had on various aspects of resource usage, as well as timing. I used a single command block to automate this workflow to rapidly implement and track my changes, which is shown below in Program 6 below.

```
make clean && \
make && \
make install && \
cd ../processor && \
make > Resources.txt 2>&1 && \
cd ../bubblesort
```

Program 6: Workflow used to synthesise and record results

This proved instrumental to a quick and iterative development cycle, allowing me to synthesise, place-and-route, and record results in a single step. This would form the basis of my workflow throughout the project.

By analysing the output text file, changes to resource utilisation could be quickly tracked to see what affect minor adjustments to the code may cause. Additionally, it allowed for comparisons with performance, enabling quick and informed decisions to be made regarding which changes would be worth pursuing.

Validation of Changes

Each change was tested using synthesis reports and simulation where possible to ensure correctness and efficiency.

The `bubblesort` program was used as the main benchmark against which changes could be compared. Due to its increased complexity compared to `softwareblink`, it exercised a broader portion of the processor's datapath and control logic, making it a more representative test case for evaluating whether changes preserved correctness. This can be seen most notably from the implementation of DSP blocks, which functioned correctly with `softwareblink` but had a minor lapse in functionality with `bubblesort`, due to the reliance in carry-on logic to perform. However, both programs were tested after every major commit to verify that basic functionality had not been erased.

For visual confirmation, the blinking patterns of LEDs were used as an effective method of identifying failures. The `softwareblink` program was expected to blink with a regular pattern, while `bubblesort` would trigger in a distinctive pattern representing the sorting algorithm.

Version Control as a Testing Tool

Git played a central role in managing and validating resource usage changes. By saving the synthesis output to a consistent `Resources.txt` file after each commit, `git diff` could be used to track even minor variations in LUT usage, RAM blocks, or timing.

When testing on hardware was not immediately possible, multiple development branches were created from a known working commit. This allowed experimental changes to be validated in isolation before being merged back into the main branch. If errors were later discovered, `git bisect` and commit-by-commit testing helped identify which change had introduced the issue, simplifying and speeding up the debugging process.

Automating synthesis and using Git made it easy to test changes quickly and reliably, without losing track of progress when changes were unsuccessfully implemented.

6 Conclusion

This project provided a comprehensive opportunity to explore low-level processor optimisation within the context of a RISC-V implementation on a resource-constrained FPGA. With a focus on reducing logic cell usage and overall resource consumption, each modification was carefully selected and evaluated for its tangible benefits in resource utilisation.

The most significant reductions were achieved through targeted removal of unused subsystems and redundant logic, such as the complete removal of the CSR subsystem. This change alone reduced block RAM usage by over 25% and freed more than 250 logic cells. Similarly, shifting the adder logic onto DSP blocks and introducing signal width trimming contributed to further improvements, freeing additional LUTs while maintaining functional integrity. In some cases, such as the signal width reduction, unexpected improvements in timing performance were also observed, indicating that some resource optimisations may bring other benefits.

A disciplined approach to development - built around modular testing, Git version control, and a systematic build and compare workflow - allowed for rapid iteration and clear attribution of each modification's impact. The `bubblesort` program served as a particularly effective benchmark, challenging the processor more thoroughly than simpler programs and ensuring that all pipeline and control logic remained operational after each change.

This experience also underscored the importance of effective team coordination and version control. While our use of branching and commit tracking improved over time, earlier adoption of formal simulation and stricter integration discipline would have likely avoided several setbacks. Limited hardware access also constrained our ability to validate changes quickly, reinforcing the need for a well-developed simulation environment from the outset.

Going forward, the freed block RAM and DSP capacity creates clear opportunities for further resource optimisation. Logic structures that currently use LUTs - such as counters - could be migrated into the available block RAM

to reduce logic cell pressure. Similarly, more arithmetic operations could be ported to DSP blocks, especially where additions or subtractions are used repeatedly. While these changes were not implemented due to time constraints, they represent practical and realistic extensions of the existing work. They would continue the theme of targeted optimisations and build upon the resource savings already achieved in this project.

References

[1] Width parameter on the FPGA *Optimizing Hardware for FPGAs*. Stitt Hub, 2023. Available at: https://stitt-hub.com/optimizing-hardware-for-fpgas/

[2] Basic CSR functionality. *Functional Safety for Control and Status Registers*. Electronic Design, 2024. Available at: https://www.electronicdesign.com

[3] Basic DSP implementation on the iCE40. *DSP Function Usage Guide for iCE40 Devices*. 2016. Available at: https://usermanual.wiki/Document/iCE40_Function_Usage_Guide.pdf

A Resource Usage Data

Modification	LUTs	Block RAMs	Global Buffers	DSPs
Baseline → PC Gating	9	0	1	0
PC Gating → Signal Width Fix	25	0	0	0
Signal Width Fix → DSP Shift	94	0	1	3
DSP Shift → CSR Removal	251	8	0	0
Total Reduction	379	8	2	3

Table 1: Individual resource reductions across each modification stage

Resource type	Used	% of total
Logic cells (`ICESTORM_LC`)	3073 / 5280	58 %
Block RAMs (`ICESTORM_RAM`)	20 / 30	66 %
Global buffers (`SB_GB`)	5 / 8	62 %
DSP blocks (`ICESTORM_DSP`)	0 / 8	0 %

Table 2: Baseline report

Resource type	Used	% of total
Logic cells (`ICESTORM_LC`)	3064 / 5280	57 %
Block RAMs (`ICESTORM_RAM`)	20 / 30	66 %
Global buffers (`SB_GB`)	6 / 8	75 %
DSP blocks (`ICESTORM_DSP`)	0 / 8	0 %

Table 3: Report after adding clock gating to the Program Counter module

Resource type	Used	% of total
Logic cells (`ICESTORM_LC`)	3039 / 5280	57 %
Block RAMs (`ICESTORM_RAM`)	20 / 30	66 %
Global buffers (`SB_GB`)	6 / 8	75 %
DSP blocks (`ICESTORM_DSP`)	0 / 8	0 %

Table 4: Report after amending the multiplexing syntax to be variable width

Resource type	Used	% of total
Logic cells (`ICESTORM_LC`)	2945 / 5280	55 %
Block RAMs (`ICESTORM_RAM`)	20 / 30	66 %
Global buffers (`SB_GB`)	7 / 8	87 %
DSP blocks (`ICESTORM_DSP`)	3 / 8	37 %

Table 5: Report after shifting addition and subtraction onto DSPs

Resource type	Used	% of total
Logic cells (`ICESTORM_LC`)	2694 / 5280	51 %
Block RAMs (`ICESTORM_RAM`)	12 / 30	40 %
Global buffers (`SB_GB`)	7 / 8	87 %
DSP blocks (`ICESTORM_DSP`)	3 / 8	37 %

Table 6: Report after removing all CSR related registers and wires

B Code

```
module mux2to1(input0, input1, select, out);
        input [31:0]    input0, input1;
        input           select;
        output [31:0]   out;

        assign out = (select) ? input1 : input0;
endmodule

module mux2to1_11bit(input0, input1, select, out);
// grep -rn "cont_mux_out" ../processor/verilog shows no usages over 11 bits
        input [10:0]    input0, input1;
        input           select;
        output [10:0]   out;

        assign out = (select) ? input1 : input0;
endmodule
```

Program 7: Fixed-width 11-bit multiplexer

```
module adder(input1, input2, is_sub, out);
    input  [31:0] input1;
    input  [31:0] input2;
    input         is_sub;
    output [31:0] out;

    wire [31:0] result;
    wire        carry_out;

    SB_MAC16 i_sbmac16 (
        .A(input1[31:16]),
        .B(input1[15:0]),
        .C(input2[31:16]),
        .D(input2[15:0]),
        .O(result),
        .CLK(1'b0),
        .CE(1'b1),
        .IRSTTOP(1'b0),
        .IRSTBOT(1'b0),
        .ORSTTOP(1'b0),
        .ORSTBOT(1'b0),
        .AHOLD(1'b0),
        .BHOLD(1'b0),
        .CHOLD(1'b0),
        .DHOLD(1'b0),
        .OHOLDTOP(1'b0),
        .OHOLDBOT(1'b0),
        .OLOADTOP(1'b0),
        .OLOADBOT(1'b0),
        .ADDSUBTOP(is_sub),
        .ADDSUBBOT(is_sub),
        .CO(carry_out),
        .CI(1'b0),
        .ACCUMCI(1'b0),
        .ACCUMCO(),
        .SIGNEXTIN(1'b0),
        .SIGNEXTOUT()
    );

    // ---- DSP Configuration using defparam ----
    defparam i_sbmac16.TOPADDSUB_LOWERINPUT         = 2'b00;  // A
    defparam i_sbmac16.TOPADDSUB_UPPERINPUT         = 1'b1;   // C
    defparam i_sbmac16.TOPADDSUB_CARRYSELECT        = 2'b11;  // Carry from BOT
    defparam i_sbmac16.BOTOUTPUT_SELECT             = 2'b00;  // Bot adder output
    defparam i_sbmac16.BOTADDSUB_LOWERINPUT         = 2'b00;  // B
    defparam i_sbmac16.BOTADDSUB_UPPERINPUT         = 1'b1;   // D
    defparam i_sbmac16.BOTADDSUB_CARRYSELECT        = 2'b00;  // No carry-in
    defparam i_sbmac16.MODE_8x8                     = 1'b0;
    defparam i_sbmac16.A_SIGNED                     = 1'b0;
    defparam i_sbmac16.B_SIGNED                     = 1'b0;

    assign out = result;
endmodule
```

Program 8: DSP implementation in adder.v

C Diagrams

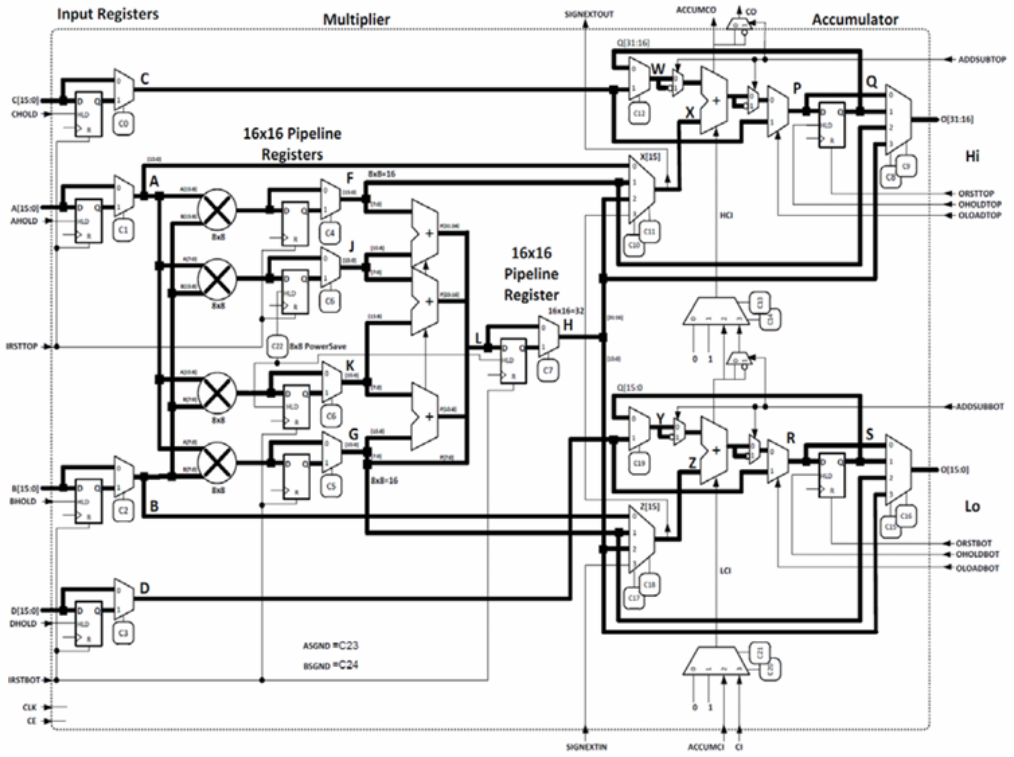

Figure 1: SB_MAC16 block structure. Source: [3]