CS2323 – Computer Architecture

Design and FPGA Implementation of 64-bit RISC-V Processors

Final Project Report

Krishna H. Patil (EE24BTECH11036)

Arnav Yadnopavit (EE24BTECH11007)

IIT Hyderabad

December 18, 2025

Contents

1 Introduction

1.1 Project motivation

From a personal standpoint, the primary motivation behind this project stemmed from a long–held aspiration to design and build a working processor. The idea of constructing a CPU from the ground up, understanding how instructions move through hardware, how data is transformed, and how control logic orchestrates every computation, has been a childhood dream. This project offered an opportunity to finally realize it.

Beyond personal ambition, we were also driven by the desire to apply the theoretical concepts learned in courses such as Digital Systems and Computer Architecture to a real, functioning hardware system. Topics such as pipelining, hazard detection, control signal generation, and memory interfacing often remain theoretical until they are encountered in actual implementations. Developing a 64-bit RISC-V processor allowed us to bridge that gap between theory and practice.

A further source of motivation was the opportunity to gain hands-on experience with FPGA development for the first time. Working with an FPGA platform provided a unique perspective on digital design, timing closure, resource constraints, and hardware debugging. Implementing a processor on the Arty A7 board offered us a practical and challenging environment to strengthen our understanding of hardware design methodologies while exploring the capabilities of modern reconfigurable logic.

Overall, this project represented the ideal convergence of personal interest, academic learning, and practical engineering experience.

1.2 Initial proposed scope

- RV64I base ISA support (integer instructions)
- Multicycle, Single-cycle, and Pipelined designs (as planned)
- RV64IM (multiply/divide) extensions
- Floating Point Unit (FPU) (planned - e.g., RV64F/RV64D)
- Cache design (e.g., direct-mapped / set-associative L1)
- UART interface for communication
- Memory-mapped I/O (LEDs, switches)
- Target FPGA: Arty A7 (or similar)

1.3 Differences Between Proposal and Final Implementation

At the beginning of the project, we proposed a broad and ambitious feature set: a multicycle or single-cycle RV64I core, a fully pipelined variant, support for the RV64IM extension, a Floating Point Unit (FPU), a cache subsystem, UART-based communication, and additional memory-mapped peripherals. As the project progressed, several of these objectives were successfully

achieved, while others required scaling down or modification due to time, complexity, and debugging challenges.

Our hardware development proceeded in three major implementation milestones. First, we completed a fully functional single-cycle RV64I processor that served as a correctness baseline. Building upon this, we implemented a 5-stage pipelined RV64I processor with hazard detection and forwarding support. Finally, we extended the pipelined design to support the RV64IM instruction set, including a pipelined and latency-aware divider unit. These three processor versions matched our core architectural goals and represented the most substantial achievements of the project.

On the peripheral side, we initially developed a working UART transmitter as part of the I/O subsystem. However, during an attempted optimization pass, a modification unintentionally broke parts of the design, and we were unable to fully restore UART functionality before the final FPGA demonstration. To compensate for this, we relied on a memory-mapped LED-based debug system that allowed us to visualize the values of registers `x0` through `x31` directly on the Arty A7 board. Although less flexible than UART output, this approach proved sufficient for hardware validation and debugging.

Some planned features,most notably the FPU and cache subsystem, remained incomplete due to their complexity and the time required for verification. Despite these deviations from the original proposal, the final implementation still delivered a robust, functioning 64-bit RISC-V processor family with meaningful peripheral integration and extensive FPGA-based testing.

2 Overall Design Overview

2.1 Development timeline followed (Phase 1, 2, 3 and 4)

Phase 1: Specification of a baseline single-cycle RV64I implementation and the creation of a testbench.

Phase 2: 5-stage pipelined implementation.

Phase 3: hazard handling, basic peripherals, RV64IM support.

Phase 4: FPGA integration, performance tuning, and demo preparation.

3 Processor Implementations

3.1 Single-Cycle RV64I Processor

The first stage of our processor development involved designing a fully functional single-cycle RV64I core. This version served as the architectural baseline for correctness before we introduced pipelining and the RV64IM extension. In a single-cycle design, all operations—fetch, decode, execute, memory access, and write-back—are completed within a single, long combinational path. While not performance-oriented, this approach significantly simplifies control and verification.

3.1.1 Datapath Architecture

The single-cycle datapath consisted of the instruction fetch block, an immediate generator, a 32-register `64-bit` register file, the ALU, branch comparison hardware, and memory access logic. All functional units are connected through combinational paths, with a single write-back point feeding the register file.

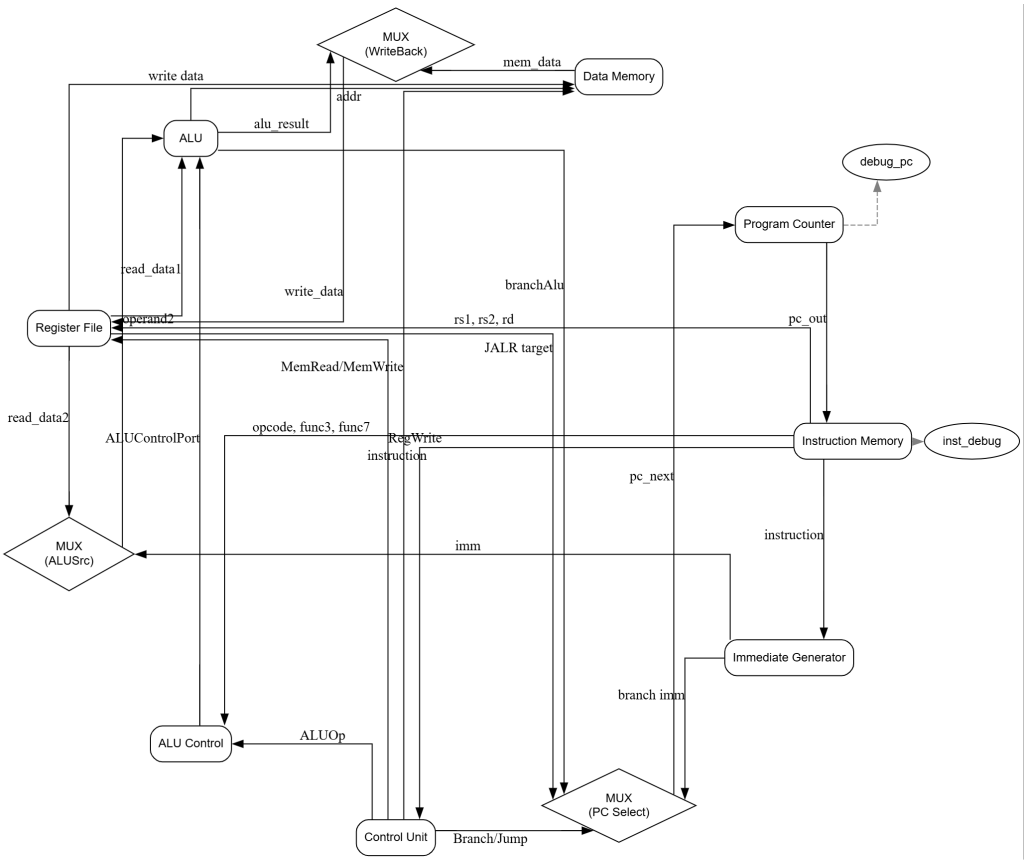

Figure 1: Single-cycle RV64I datapath

Because the entire instruction executes in one clock cycle, there is no need for hazard management or pipeline stalls. The simplicity of this architecture made it ideal as a first implementation phase.

3.1.2 Control Logic

The control logic was implemented as a purely combinational decoder. Based on the opcode and funct3/funct7 fields, the controller generated signals for ALU operation selection, branching decisions, register write enables, memory read/write enables, and immediate type decoding. The absence of pipelining meant the control unit did not need stall, forwarding, or branching prediction mechanisms.

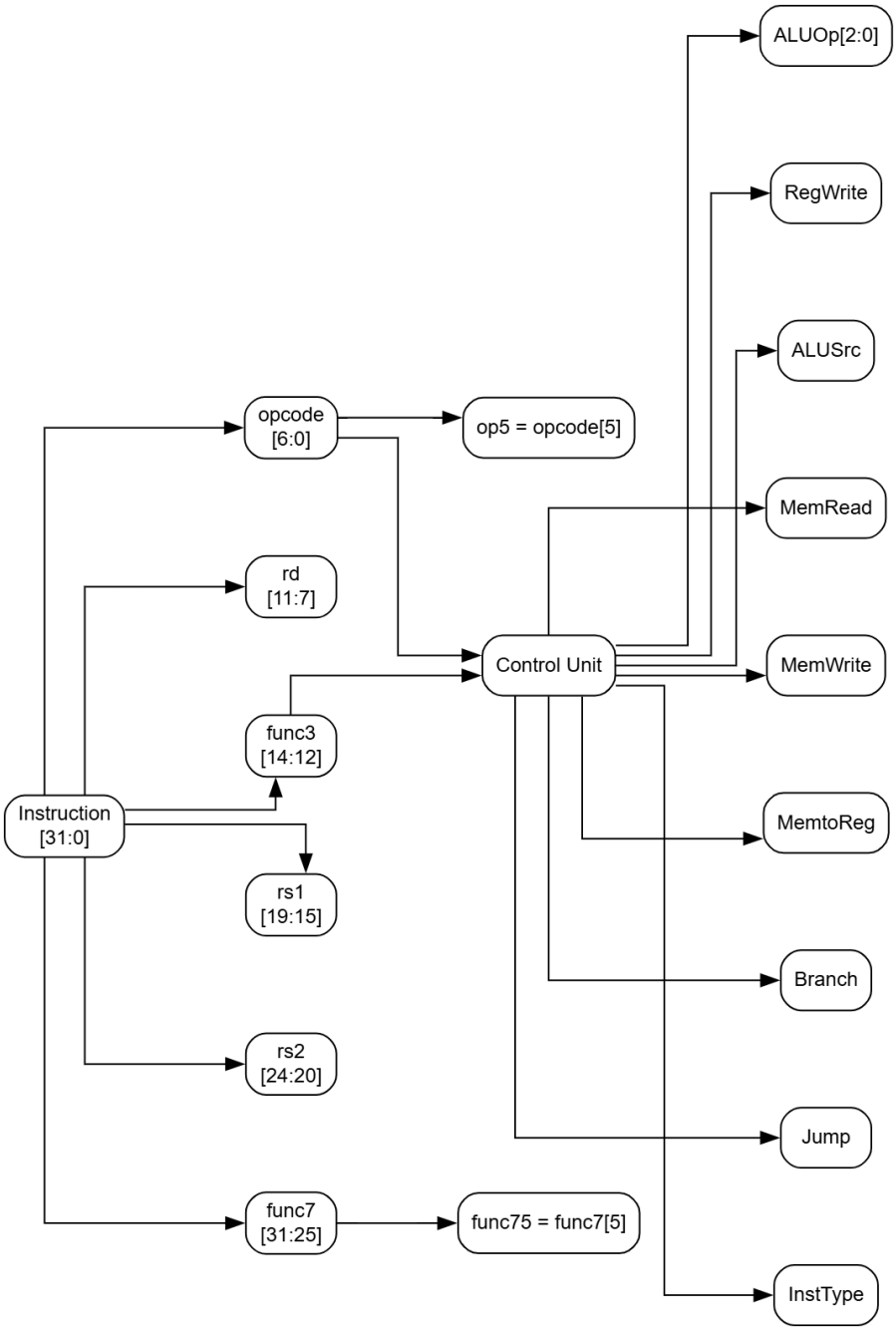

Figure 2: Single-cycle control logic diagram

3.1.3 Supported Instructions

The implementation covered the complete RV64I base instruction set. All arithmetic, logical, branch, load/store, and control-flow instructions were supported. The ALU included operations such as `ADD`, `SUB`, `AND`, `OR`, `XOR`, comparison operations, shift operations, and branch target calculations.

3.1.4 Simulation and Testing

To test the single-cycle core, we wrote multiple assembly programs. Since we did not use the RISC-V GNU toolchain, we relied on two online tools:

- An online C-to-assembly converter for generating RISC-V assembly from small C programs.
- An online assembly-to-machine-code converter for producing executable hexadecimal machine code.

To integrate these machine instructions into our FPGA workflow, we developed a small Python script to automatically adjust formatting and alignment so the resulting machine code could be directly pasted into the `inst.coe` file for Block RAM initialization. This automation allowed rapid iteration during testing.

Our test programs included:

- A division routine (software-implemented division loop).
- A bubble sort implementation operating on an array in memory.
- A GCD (Greatest Common Divisor) algorithm using the Euclidean method.

These programs verified arithmetic correctness, memory access behavior, control flow operations, and overall datapath stability. Waveform inspection through simulation tools allowed us to confirm instruction-level correctness before moving to pipelined versions of the processor.

3.2 5-Stage Pipelined RV64I Processor

After validating the single-cycle core, we extended the design into a 5-stage pipelined RV64I processor. The pipelined architecture significantly improved throughput by overlapping instruction execution across the IF, ID, EX, MEM, and WB stages. Pipeline registers separated each stage, allowing multiple instructions to be in-flight simultaneously.

3.2.1 Instruction Fetch (IF)

The IF stage computed the Program Counter (PC) and fetched instructions from instruction memory. Since branch resolution was moved earlier into the decode stage, the PC selection logic depended on signals coming directly from ID. The IF stage was also equipped with stall inputs so it could freeze during division stalls or memory stalls.

3.2.2 Instruction Decode (ID)

The ID stage decoded instructions, read source registers, and generated immediates. A major architectural improvement was relocating branch comparison from the EX stage to the ID stage. This was implemented through a dedicated `Branch_D` module responsible for evaluating branch conditions (BEQ, BNE, BLT, BGE, etc.) using the RS1 and RS2 operands.

This reduced branch penalty from two flushed stages to one, since the decision was made earlier. The ID stage also handled hazard detection and generating stall signals for load-use hazards, division stalls, and data memory stalls.

3.2.3 Execute (EX)

The EX stage performed ALU operations such as arithmetic, shifts, and logical operations. Since branch decisions were already resolved in the ID stage, the ALU no longer computed branch conditions. Instead, it focused purely on computation and calculating memory addresses for load/store instructions.

Pipeline forwarding from later stages (EX/MEM and MEM/WB) fed the ALU inputs when necessary.

3.2.4 Memory Access (MEM)

The MEM stage communicated with data memory. In addition to standard load/store functionality, this stage included handshake and busy signals for the BRAM-based memory system. If the memory unit asserted a `dmem_stall`, the pipeline froze until memory became ready. This mechanism simplified timing but required tight integration with the global stall controller.

3.2.5 Write Back (WB)

The WB stage wrote the appropriate ALU or memory result back to the register file. A simple multiplexer selected between ALU output and memory-read data. This completed the instruction execution flow.

3.2.6 Hazard Detection Unit

The hazard detection unit monitored dependencies between instructions in ID and EX. Classic load-use hazards triggered a one-cycle stall. Additionally, global stall signals from the divider unit and data memory system were integrated into this module. When a stall was asserted, the IF and ID stages froze, and EX received a bubble.

3.2.7 Forwarding Unit

The forwarding unit minimized stalls by routing results from the EX/MEM or MEM/WB pipeline stages directly into the ALU input multiplexers. It supported all arithmetic and logical instructions, enabling back-to-back dependent instructions. Load-use hazards and multicycle division operations were the only cases requiring explicit stalls.

3.2.8 Branch Handling (Using Branch_D Module)

Unlike typical pipelines where branches are resolved in EX, we implemented early branch detection in the decode stage through a dedicated `Branch_D` module. It performed:

- Operand comparison using RS1 and RS2,
- Generation of branch-taken signals,
- Calculation of immediate-based branch targets,
- Flushing the instruction in IF when required.

This significantly reduced branch penalties and simplified EX-stage logic.

3.2.9 Pipeline Diagram

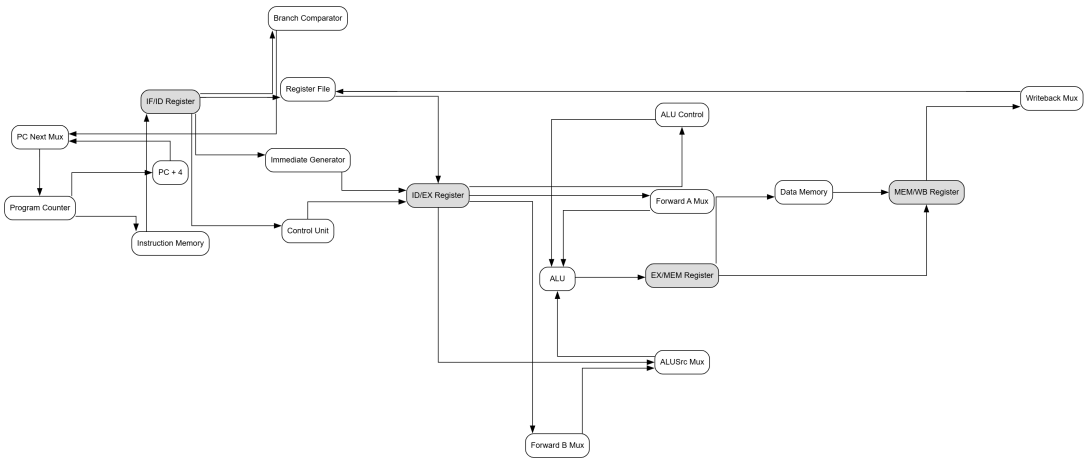

Figure 3: 5-stage pipeline diagram with stalls and early-branch detection.

3.3 RV64IM Pipelined Processor

After completing the RV64I pipelined processor, the next step was extending the design to support the full RV64IM instruction set. This required integrating hardware support for multiplication and division. While multiplication was mapped cleanly into the ALU datapath as a single-cycle combinational operation, division required a separate multicycle pipelined unit. This introduced new control complexities, particularly around stalling the pipeline correctly while the divider was busy.

3.3.1 Single-Cycle Multiply Unit

All multiply-class instructions (`MUL`, `MULH`, `MULHSU`, `MULHU`) were implemented inside the EX-stage ALU as purely combinational operations. The Artix-7 DSP slices allowed 64-bit signed multiplication to be performed in a single cycle. As a result:

- no special stall logic was required,
- existing forwarding paths handled dependency resolution,
- no changes were required in pipeline control.

3.3.2 Pipelined Divider (Xilinx div_gen IP)

Division (`DIV`, `DIVU`, `REM`, `REMU`) was implemented using the Xilinx `div_gen` IP configured as a Radix-2 pipelined divider with:

- 64-bit signed dividend,
- 64-bit signed divisor,
- 128-bit output,
- a fixed latency of 7 cycles.

The divider exposed two key signals:

- `div_busy`: asserted while the divider is computing,
- `div_done`: asserted when the output is valid.

3.3.3 DivStaller Module

Because division instructions occupy the EX stage for multiple cycles, a dedicated `DivStaller` module was created. It contains an FSM-based counter initialized to the divider latency. Its outputs are:

- **DivStalled**: high while the divider is computing,
- **Divreset**: a one-cycle pulse used to reset or trigger the divider core.

During **DivStalled**:

- IF is frozen (`StallF`),
- ID is frozen (`StallD`),
- EX holds the division instruction,
- EX/MEM receives a bubble,
- WB is protected to avoid early write-back.

3.3.4 Data Memory Stallers

Loads and stores used synchronous BRAM, which may introduce delay depending on read/write operations. The `dmemstaller` module produced a `MemStall` signal when the memory was busy.

When `MemStall` is asserted:

- IF and ID freeze,
- EX is bubbled once,
- MEM holds the current load/store.

3.3.5 Interaction with Hazard Detection and Forwarding

The hazard detection unit was extended to accept:

- `DivStalled`, and
- `MemStall`

as additional stall sources. These operate alongside classic load-use hazard detection. Forwarding was unchanged except that divider results are forwarded only when `div_done` is asserted.

3.3.6 Pipeline Stage Behavior for RV64IM Instructions

The introduction of multicycle division changed the behavior of several pipeline stages:

IF Stage

- Stalled whenever `DivStalled` or `MemStall` is high.
- PC does not update during a division.

ID Stage

- Also frozen on divider or memory stalls.
- Branch resolution and branch target calculation still occur during ID, unaffected by the divider.

EX Stage

- For multiply instructions: behaves like a normal ALU cycle.
- For divide instructions:
  - The division instruction remains in EX for the full divider latency.
  - The EX/MEM register receives bubbles while division is in progress.
  - The divider receives a one-cycle `Divreset` pulse at start.

MEM Stage

- Receives bubbles for division instructions.
- May independently stall due to synchronous BRAM using `MemStall`.

WB Stage

- Multiply results write back normally after one cycle.
- Division results write back only in the cycle where `div_done` is asserted.
- Forwarding from the divider is permitted only when valid.

3.3.7 Testing Strategy

Testing the RV64IM implementation relied on both our previous RV64I validation programs and new tests designed specifically for the M-extension. Since the underlying pipeline, forwarding logic, and branch handling remained the same, all earlier programs (division routine, bubble sort, GCD, arithmetic tests, memory tests, and branching programs) were rerun to ensure that adding the M-extension did not break RV64I functionality.

In addition to these regression tests, we created a dedicated "all-instruction run" testbench. This testbench executed at least one instance of every instruction we implemented—including

all ALU operations, branches, jumps, loads/stores, multiply variants, and divide/remainder variants. The goal of this testbench was to ensure that:

- every pipeline path was exercised,
- hazard detection and forwarding worked correctly under M-extension interactions,
- division stalls and memory stalls coexisted correctly with branching and load-use hazards,
- results matched software-generated reference outputs.

For generating programs, we continued relying on online C-to-assembly and assembly-to-machine-code tools. A Python script automatically formatted and aligned the resulting machine code so that it could be pasted directly into the `inst.coe` file for Block RAM initialization.

Waveform-level verification was performed extensively to monitor:

- the `Divreset` pulse,
- the `DivStalled` duration,
- correct bubbling of EX/MEM during division,
- correct write-back timing when `div_done` asserted,
- correct forwarding behavior during multiply and divide sequences.

This combination of all-instruction testing, and waveform inspection gave us high confidence that the RV64IM pipeline operated correctly.

Overall, the RV64IM integration required significant architectural modifications, particularly around EX-stage stalls. However, the use of independent staller modules (rather than a unified stall controller) resulted in a modular and easily debuggable pipeline.

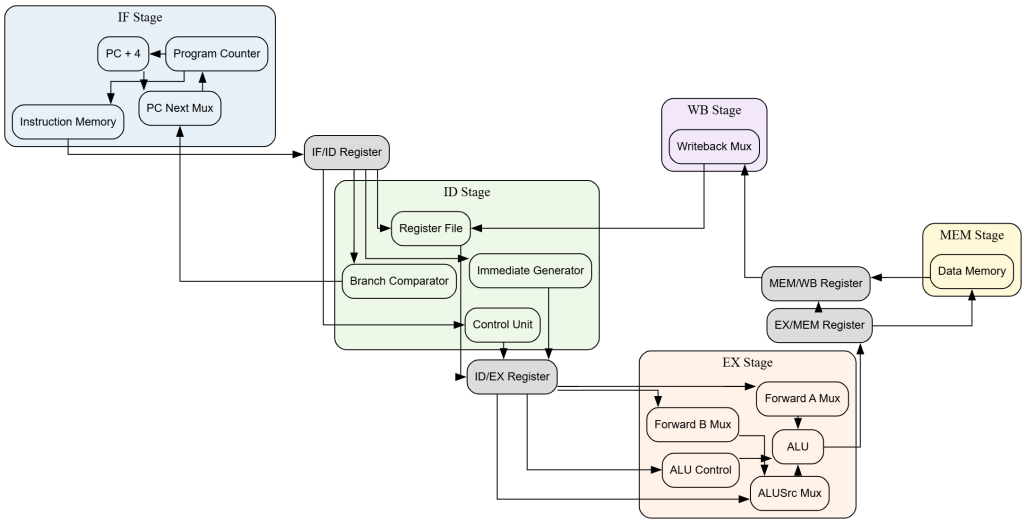

Figure 4: RV64IM pipeline

4 Unfinished / Modified Features

4.1 Floating Point Unit (FPU)

Although an F-extension floating-point unit was part of our original project proposal, we were ultimately unable to complete it within the project timeframe. Several factors contributed to this:

- **Time limitations:** The integration of the RV64IM pipeline itself required significantly more time than expected, especially for handling multicycle division, stalling behavior, and memory interactions. By the time the RV64IM core became stable, the remaining time was insufficient for a full FPU implementation.
- **Complex design decisions:** Implementing a fully IEEE-754-compliant FPU requires several architectural choices that we could not finalize in time:
  - determining the number of pipeline stages required for floating-point add, subtract, multiply, and divide,
  - selecting between a fully pipelined or iterative design for floating-point operations,
  - handling rounding modes, exceptions, and denormal numbers,
  - deciding the latency of each operation and how these latencies would affect pipeline stalls.
- **Register file integration challenges:** Floating-point instructions require a dedicated 32-entry floating-point register file (`f0`–`f31`). We needed to determine:
  - how to integrate this F-register file alongside the standard integer register file,
  - how to route operands between the FPU and the main datapath,
  - how to manage write-back arbitration between floating-point and integer results,
  - how to handle forwarding and hazards involving both integer and floating-point pipelines.
- **Pipeline interaction complexity:** Integrating the FPU with our existing pipeline would have required:
  - defining stall behavior for long-latency floating-point operations,
  - extending the Hazard Detection Unit to support floating-point hazards,
  - updating the forwarding network to handle FPU outputs,
  - modifying the decode stage to decode floating-point opcodes and format F-type immediates.
- **Communication/interface uncertainties:** We had not finalized how the FPU would exchange data and control signals with the main processor pipeline. Questions such as whether FP results would bypass directly into EX or return via a dedicated write-back path were still unresolved.

Given these architectural complexities and the limited remaining time, implementing the FPU would have jeopardized the stability of the entire processor pipeline. For these reasons, we made the deliberate decision to postpone the FPU to future work, ensuring that our RV64IM implementation remained correct and robust.

4.1.1 Cache Design — Planned vs Actual

Our initial project proposal included designing and integrating a simple instruction and data cache to improve memory performance. However, as development progressed, we realized that caching did not make sense within the constraints of our current memory architecture.

In our implementation, both the instruction memory (IMEM) and data memory (DMEM) were mapped directly onto the FPGA's on-chip Block RAM (BRAM). BRAM provides:

- single-cycle access latency,
- no need for burst transfers or prefetching.

Under these conditions, a cache would not provide any performance improvement. Since BRAM is already faster than a realistic cache hierarchy, introducing caching would only add unnecessary complexity without any measurable benefit.

To meaningfully justify a cache, we would need to relocate our memories to the external DDR3 RAM available on the Arty A7 board. Only then would realistic off-chip memory latencies (tens of cycles) make caching beneficial. However:

- integrating DDR3 would require using the MIG (Memory Interface Generator) IP,
- the MIG adds significant design complexity and requires careful timing closure,
- adapting our RV64IM pipeline to handle DDR3's burst-based and multi-cycle latency model would require extensive redesign of the memory subsystem.

Due to these practical constraints and time limitations, we decided to postpone cache implementation. The idea remains part of our planned future work, but for this project, BRAM-based IMEM and DMEM made caching unnecessary and impractical.

4.2 UART Interface

The UART interface was originally intended to provide a convenient way to observe register and memory contents during FPGA execution using a serial terminal. Our initial implementation focused on UART transmission, and during early testing phases it worked reliably. However, later design changes led to unexpected failures that we were not able to resolve before the final presentation.

Initial Working Design The first version of the UART subsystem consisted of:

- a `uart_tx` module for serial transmission,
- a custom `uart_streamer` module to serialize multi-byte words,

- a `dump_controller`, a FSMresponsible for formatting debug data for output,
- a `halt_detector` that triggered the data dump once the program halted.

In this setup, we used a "spare" internal debug signal named `debug_regfile`, which exposed the contents of the integer register file. Using a simple generate block, we copied the register values into this debug bus whenever needed. The UART streamer then transmitted the data byte-by-byte to PuTTY, allowing us to view all 32 registers directly on the terminal. This system worked flawlessly and became our primary debugging tool during early development.

Attempt to Add Memory Dump Support Encouraged by the success of register dumping, we attempted to extend the UART system to also dump the contents of data memory. Our plan was to:

- integrate the DMEM read interface into the dump controller,
- sequentially stream memory bytes just like the register file,
- display the entire memory state on PuTTY after program halt.

However, after integrating memory dumping logic, the UART system suddenly stopped working. The transmitter no longer produced valid output, and the dump controller appeared to freeze or misalign the outgoing data stream. Despite extensive debugging—checking timing, stalling behavior, FSM state transitions, and reset logic—we were unable to isolate the cause of the failure.

This issue persisted until the final hours before the presentation, and due to time constraints we ultimately could not restore full UART functionality.

UART Receiver (Unfinished) We also implemented a `uart_rx` module intended to accept program input from the user and dynamically populate `inst.coe`. While the receiver module itself functioned in simulation, we did not have enough time to design the infrastructure needed to:

- buffer incoming bytes,
- decode ASCII/hex formats,
- write them reliably into instruction memory on the FPGA.

This feature remains unfinished.

Final Workaround: LED-Based Debug Output To ensure we could still demonstrate correct processor behavior on hardware, we relied on the "spare" LED-based debug system we had built earlier. The Arty A7 board provides:

- 4 standard LEDs,
- 4 RGB LEDs,

- 4 switches,
- multiple push buttons.

We used the switches to select a register index (0–15), and a push button acted as a "fifth bit" to access registers 16–31. The LEDs displayed the chosen register's value in a simplified pattern. Although limited, this system allowed us to meaningfully demonstrate register values on real hardware without relying on UART output.

In summary, the UART interface was initially functional and extremely useful for debugging. However, our attempt to expand it with memory-dump capability introduced unintended side effects that could not be resolved within the time available. The fully featured UART system remains an important target for future work.

5 Results

This section summarizes the synthesis, timing, resource utilization, power results, and real FPGA-tested operating frequencies for all three versions of our processor.

5.1 Timing Results (Post-Implementation WNS)

The post-implementation Worst Negative Slack (WNS) and corresponding theoretical maximum clock frequencies are shown in Table 1.

Table 1: Timing Results (WNS) and Estimated Maximum Clock Frequency

Design	**WNS (ns)**	**Estimated $f_{\max}$ (MHz)**
RV64I Single-Cycle	36.349 ns	$\approx$ 27.5 MHz
RV64I 5-Stage Pipeline	12.162 ns	$\approx$ 82.2 MHz
RV64IM 5-Stage Pipeline	52.618 ns	$\approx$ 19.0 MHz

5.2 Experimentally Verified Clock Speeds

Although the theoretical $f_{\max}$ estimates give an upper bound, we also tested each design directly on the Arty A7 FPGA.

Table 2: Experimentally Verified Stable Clock Frequencies

Design	**Tested Frequency (MHz)**	**Result**
RV64I Single-Cycle	25 MHz	Stable
RV64I 5-Stage Pipeline	50 MHz	Stable
RV64IM 5-Stage Pipeline	15 MHz	Stable
RV64IM 5-Stage Pipeline	18 MHz	Functional but timing-sensitive

The RV64IM design can run up to **10 MHz reliably**, and up to **15 MHz with caution**. For safety and to guarantee timing closure, we recommend running it at **5 MHz** during demonstrations.

The RV64I pipelined processor demonstrated excellent timing and remained stable at **50 MHz**, confirming the benefit of pipelining. The single-cycle design operated correctly up to **25 MHz**, as expected for a long-combinational-path architecture.

5.3 Resource Utilization

Table 3: FPGA Resource Utilization Summary

Design	LUT	FF	BRAM	DSP
RV64I Single-Cycle	2983 / 2931	2048	1.5	4
RV64I 5-Stage Pipeline	3413 / 3370	2618	32	6
RV64IM 5-Stage Pipeline	13900 / 13462	13475 / 5277	32	54

The sharp increase in DSP usage for the RV64IM design stems from the 64-bit multiplier and the Xilinx Divider IP.

5.4 Code Size Comparison

Table 4: RTL Code Size Comparison

Design	Lines of Verilog
RV64I Single-Cycle	510
RV64I 5-Stage Pipeline	993
RV64IM 5-Stage Pipeline	1273

5.5 Power Analysis

Table 5: Estimated Dynamic Power Consumption

Design	Power (W)
RV64I Single-Cycle	0.093 W
RV64I 5-Stage Pipeline	0.094 W
RV64IM 5-Stage Pipeline	0.103 W

6 tb results

```
================================================
Register File Check at 990ns
================================================
x0 = 0   PASS
x1 = 5   PASS
x2 = 7   PASS
x3 = 18446744073709551613   PASS
x4 = 2   PASS
x5 = 0   PASS
x6 = 12   PASS
x7 = 2   PASS
x8 = 5   PASS
x9 = 7   PASS
x10 = 2   PASS
x11 = 20   PASS
x12 = 2   PASS
x13 = 18446744073709551615   PASS
x14 = 15   PASS
x15 = 1   PASS
x16 = 13   PASS
x17 = 9   PASS
x18 = 1   PASS
x19 = 0   PASS
x20 = 12   PASS
x21 = 19   PASS
x22 = 24   PASS
x23 = 5   PASS
x24 = 12   PASS
x25 = 12   PASS
x26 = 1   PASS
x27 = 2   PASS
x28 = 144   PASS
x29 = 9   PASS
x30 = 256   PASS
x31 = 0   PASS
------------------------------------------------
Total PASS: 32, FAIL: 0
 ALL TESTS PASSED
================================================
```

Figure 5

```
==========================================
Register File Check at 990ns
==========================================
x0 = 0  PASS
x1 = 15  PASS
x2 = 18446744073709551613  PASS
x3 = 200  PASS
x4 = 18446744073709551571  PASS
x5 = 18446744073709551615  PASS
x6 = 18446744073709551615  PASS
x7 = 0  PASS
x8 = 18446744073709551611  PASS
x9 = 13  PASS
x10 = 0  PASS
x11 = 0  PASS
x12 = 0  PASS
x13 = 5  PASS
x14 = 18446744073709551613  PASS
x15 = 15  PASS
x16 = 0  PASS
x17 = 0  PASS
x18 = 0  PASS
x19 = 0  PASS
x20 = 0  PASS
x21 = 0  PASS
x22 = 0  PASS
x23 = 0  PASS
x24 = 0  PASS
x25 = 0  PASS
x26 = 0  PASS
x27 = 0  PASS
x28 = 0  PASS
x29 = 0  PASS
x30 = 0  PASS
x31 = 0  PASS
------------------------------------------
Total PASS: 32, FAIL: 0
 ALL TESTS PASSED - M Extension Functional!
==========================================
```


Figure 6: RV64IM pipeline

```
==========================================
   Register File Values for x27 to x31
==========================================
x27 = 1
x28 = 3
x29 = 5
x30 = 7
x31 = 9
==========================================
```

Figure 7: Bubble Sort

6.1 Discussion

The results illustrate the architectural trade-offs clearly:

- **Single-cycle RV64I:** Simple and small, but limited by long combinational paths.
- **RV64I pipeline:** Excellent frequency and efficiency; best performance per LUT.
- **RV64IM pipeline:** Most feature-rich, but slowed by the single-cycle 64-bit multiplier and multicycle Divider IP.

Overall, each stage of our processor shows clear progression in capability, complexity, and performance.

6.2 Future Work

Several extensions and architectural improvements remain open for future development. Completing these would significantly enhance the capability, performance, and usability of our processor:

- **Full FPU Integration (RV64IMF):** Completing a fully IEEE-754 compliant floating-point unit remains one of our primary goals. This includes implementing FP add, subtract, multiply, divide, rounding modes, exception flags, and an independent floating-point register file. We also aim to integrate the FPU cleanly into the existing pipeline without excessive stalls.
- **Cache Hierarchy (L1 I-Cache and D-Cache):** A realistic cache subsystem will be needed once IMEM/DMEM are moved to external DDR3 memory. Planned features include a direct-mapped or 2-way associative L1 cache, write-back policy, and miss handling FSMs.
- **Fixing and Optimizing the UART Subsystem:** Restoring the previously working UART TX, completing the UART RX pathway, and building a reliable memory/register dump interface are high-priority tasks. A simple binary protocol could greatly improve debugging usability.

- **Improved I/O Support:** We aim to integrate a small display or OLED screen, along with more structured GPIO/I/O peripherals, enabling high-quality on-board interaction without depending on serial terminals.
- **Enhanced Verification Framework:** Future revisions will incorporate stronger verification techniques, including:
  - formal property checking ,
  - constrained and formal testbenches,
  - automated instruction-level testing.
- **Achieving Higher Clock Frequencies:** We are determined to push the RV64IM (and future RV64IMF) core toward a stable operating frequency of **50 MHz or more**. This will require:
  - rebalancing pipeline stages and adding more stages if needed ,
  - optimizing the multiplier and divider paths,
  - reducing long fanout nets,
  - improving placement and routing constraints.

  Our goal is to outperform typical microcontroller-class CPUs (such as Arduino-class cores) by a wide margin.
- **Long-Term Architectural Ambitions:** As a personal objective, future exploration includes extending this project toward more advanced CPU architectures such as:
  - a basic VLIW processor
  - a simple superscalar pipeline,
  - or even a multicore RISC-V system-on-chip.

In summary, our processor forms a strong foundation for continued learning, experimentation, and architectural exploration.

6.2.1 Challenges for Future Extensions

While our processor forms a strong foundation for additional architectural features, several challenges make further development significantly more demanding. First, implementing large extensions such as a full IEEE-754 FPU, cache hierarchy, improved I/O subsystems, or display interfaces is not realistically achievable by a team of only two students. Each new component—whether an FPU pipeline, UART subsystem, cache controller, or display driver,requires careful integration with all existing pipeline stages. Ensuring that these independently designed modules work together without causing timing violations, structural hazards, or interface mismatches becomes increasingly time-consuming and complex.

In particular, extending the design with a floating-point unit demands new multi-stage pipelines, precise stall and forwarding logic, and extensive timing optimization. Achieving clean

timing closure for such long-latency units on the Artix-7 requires careful microarchitectural planning and iterative experimentation.

Similarly, adding richer I/O functionality requires proper interfacing protocols, handshake mechanisms, buffering strategies, and reliable clock-domain crossing when necessary. For display-based output, an additional challenge is converting raw register or memory values into human-readable characters, which requires implementing binary-to-ASCII conversion hardware or a small formatted output processor.

These challenges highlight that as the processor grows in complexity, so does the engineering effort needed for correct integration, verification, and timing optimization. Future extensions will therefore require not only more manpower but also more disciplined design methodology and more advanced verification infrastructure.

Project Files and Resources

All project files, RTL code, testbenches, IP configurations, simulation outputs, and documentation are available on our project Google Drive:

https://drive.google.com/drive/folders/1p9z1YLM3DZuk39lXkIerJmVoPrgTB8Wx?usp=sharing

References

- Patterson & Hennessy (Hennessy and Patterson, Computer Organization and Design, RISC-V edition).
- Digital Design and Computer Architecture by Sarah L. Harris and David Harris
- RISC-V Foundation. RISC-V Instruction Set Manual.
- Xilinx / Digilent FPGA board documentation (Arty A7).

Appendix: README and Usage Guide

A. Steps for Running an Assembly (`.s`) File

*Note: For detailed IP configuration screenshots (IMEM, DMEM, Divider), refer to the Appendix section titled **"IP Generation Settings."***

1. Obtain the machine code (hex) for your assembly program using the online RISC-V assembler:
   https://riscvasm.lucasteske.dev/
2. Copy the generated hex instructions into the `instructions.py` script in the `instructions/` directory.
3. Run the Python script to generate formatted contents compatible with Vivado's `inst.coe` file.
4. Paste the formatted output into the appropriate section of `inst.coe`.
5. Close and reopen the Vivado project to ensure the Block RAM is reinitialized, then run the simulation.
6. **Important:** Ensure that `testb_dumpreg.v` (or the appropriate testbench) is set as the simulation top.

B. Setting Up the RISC-V Processor for the First Time

*Note: IP configuration examples for IMEM, DMEM, and Divider IP can be found in the Appendix under **"IP Generation Settings."***

1. Unzip the project directory and open Vivado.
2. When creating a new project, choose **Add Existing Sources**, and select one RTL folder inside `RISCV-Processor/rtl/`, such as:
   - `Single_Cycle/`
   - `Pipelined_CPU/`
   - `Pipelined_CPU_64I/`
3. Right-click `imem_ip` → `Re-customize` → `Other Options` and browse to select the correct `inst.coe` file for initialization.
4. Click **Finish**, then generate output products in **Global mode** (not "Out-of-context").
5. Generate the remaining IPs as required:
   - `dmem_ip` in `dmem_top.v`
   - `div_gen_0` in `ALU64.v` (only needed for the RV64IM design)
6. Add the required testbenches under `testcodes/`. The `dump_reg` testbench is typically sufficient for most programs.

Appendix: IP Generation Settings

This appendix contains the IP configuration settings for regenerating the IMEM, DMEM, and Divider IP used in our processor design.

A. Instruction Memory (IMEM) — Block Memory Generator

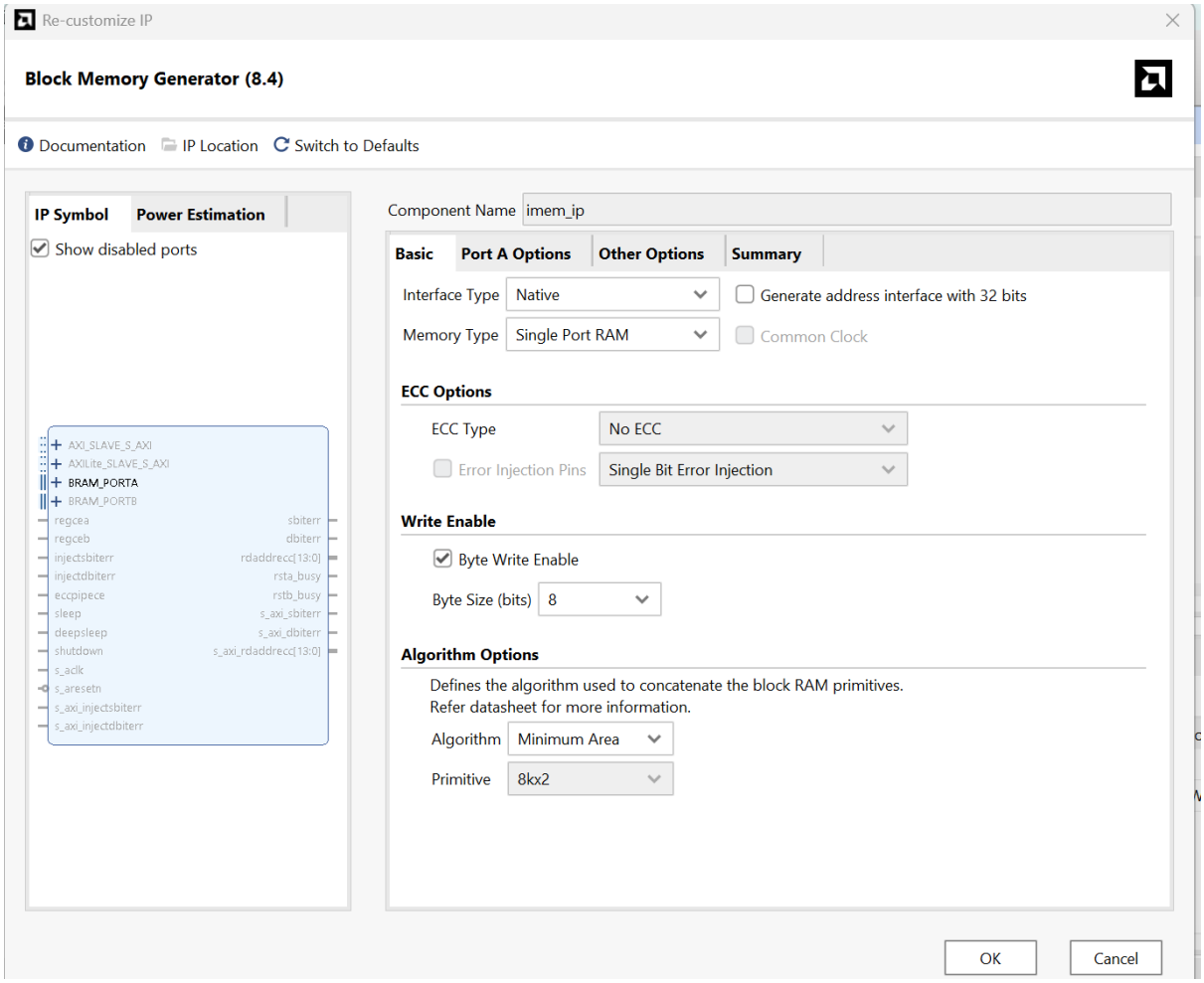

Figure 8: IMEM IP — Basic Configuration

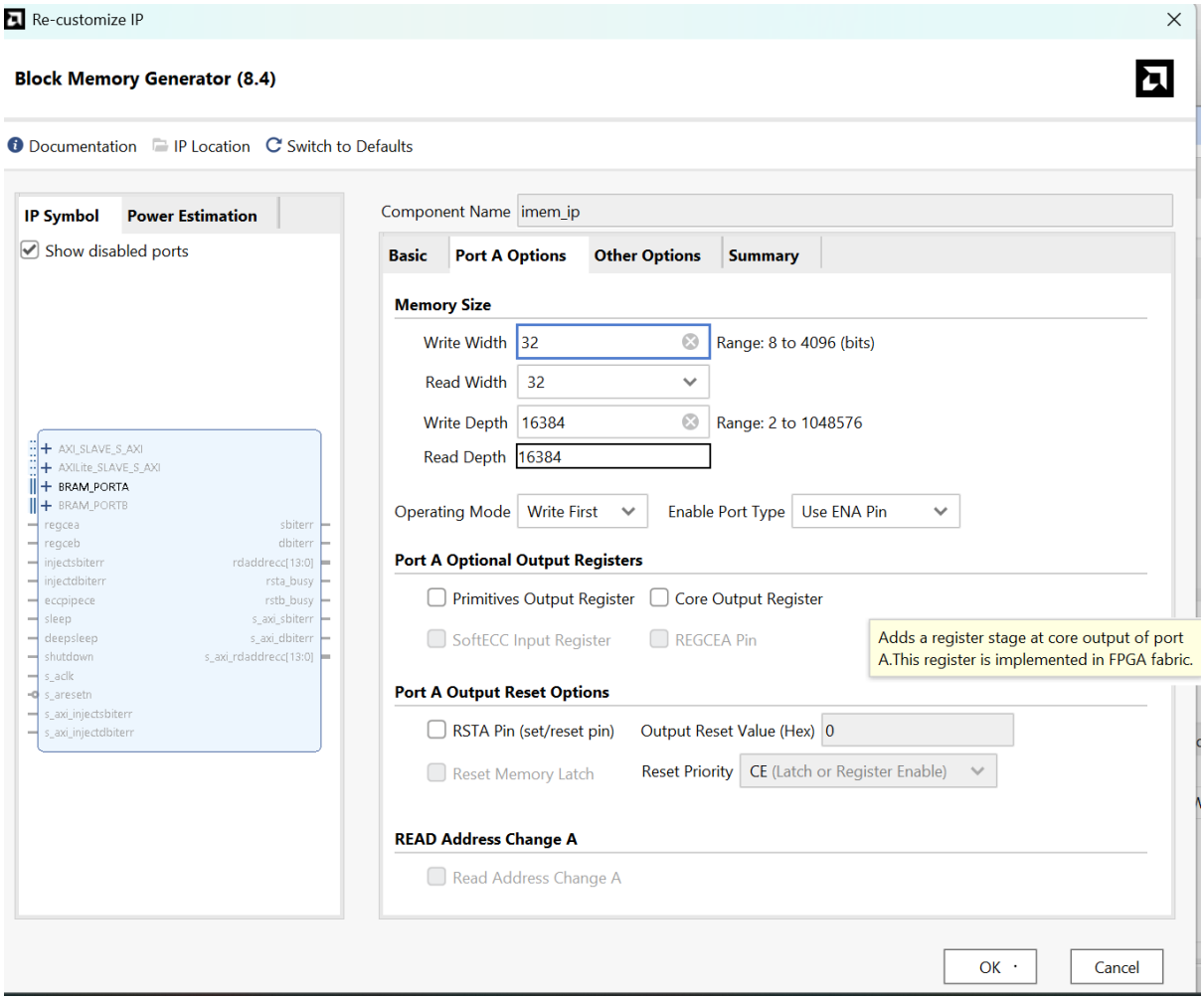

Figure 9: IMEM IP — Port A Options

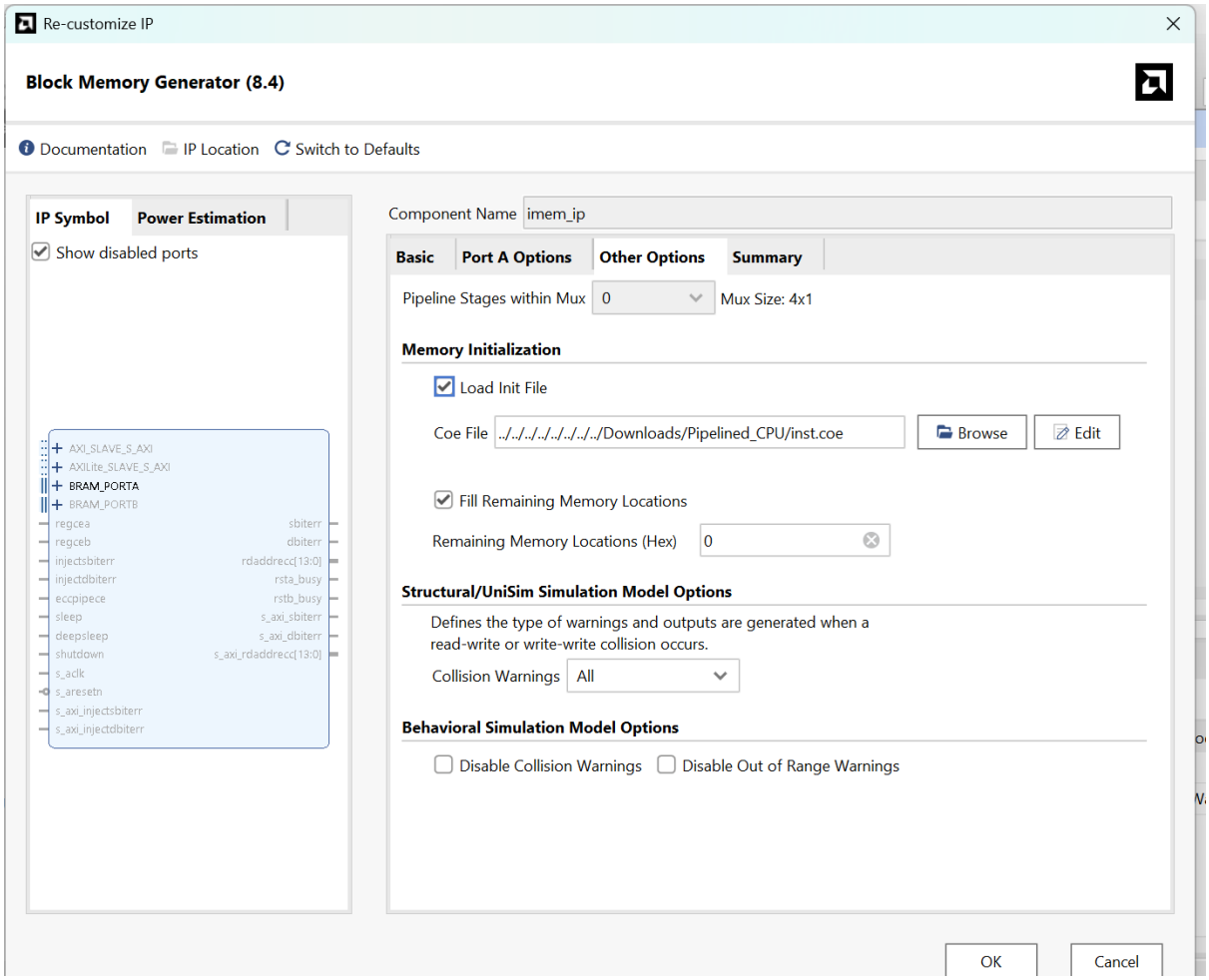

Figure 10: IMEM IP — other Options

B. Data Memory (DMEM) — Block Memory Generator

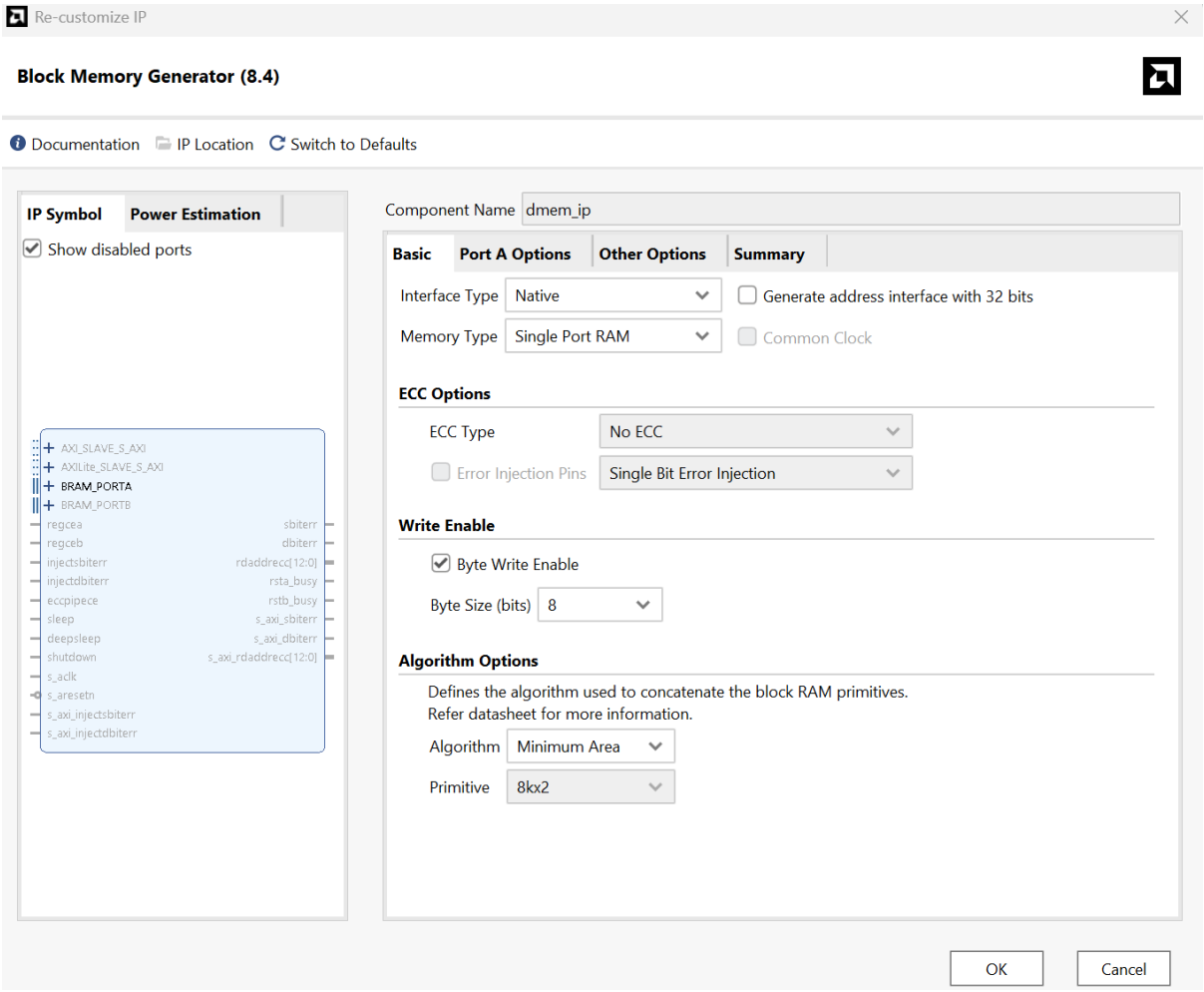

Figure 11: DMEM IP — Basic Configuration

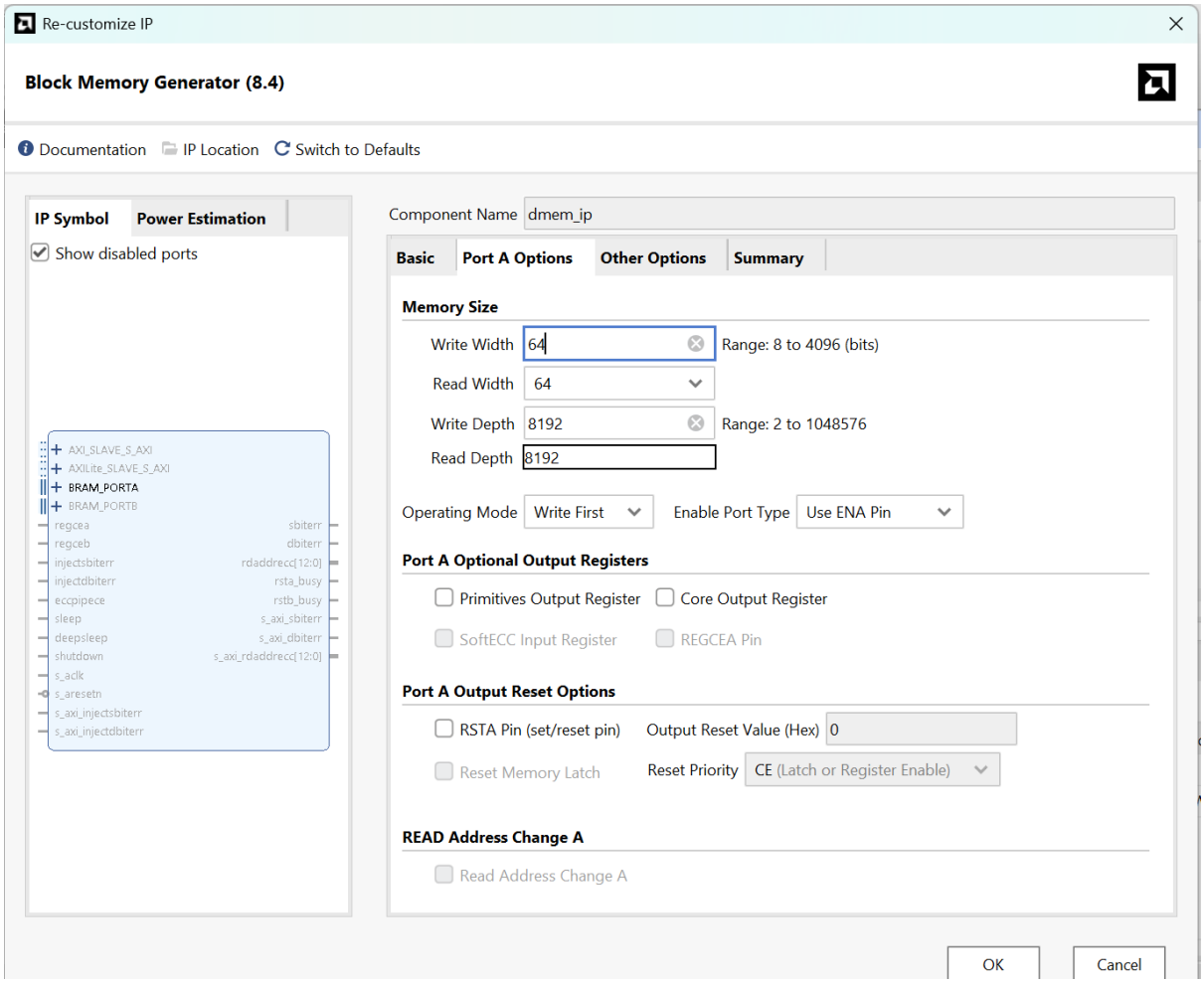

Figure 12: DMEM IP — Port A Options

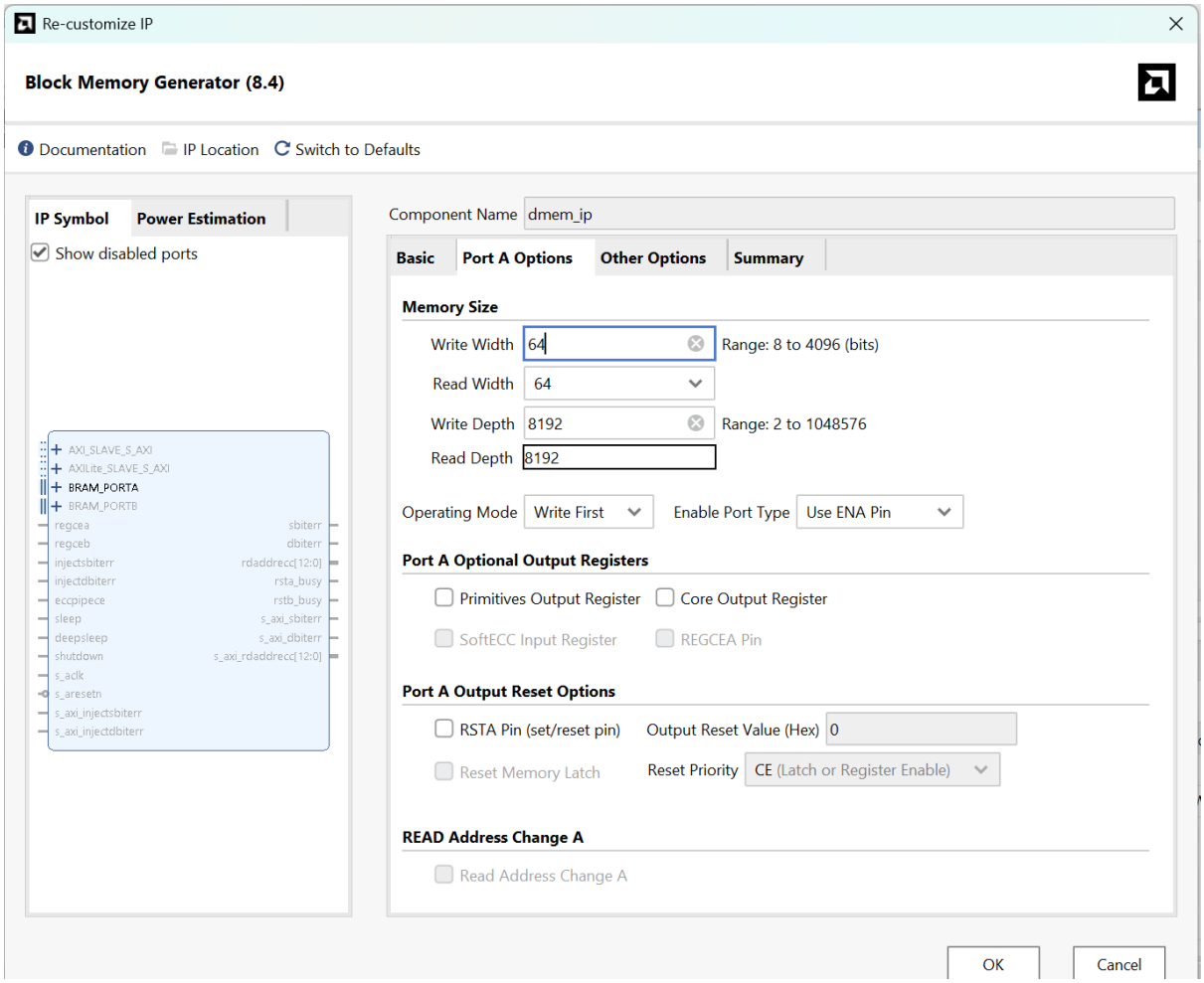

Figure 13: DMEM IP — Other Options

C. Divider Generator IP (RV64IM Division Unit)

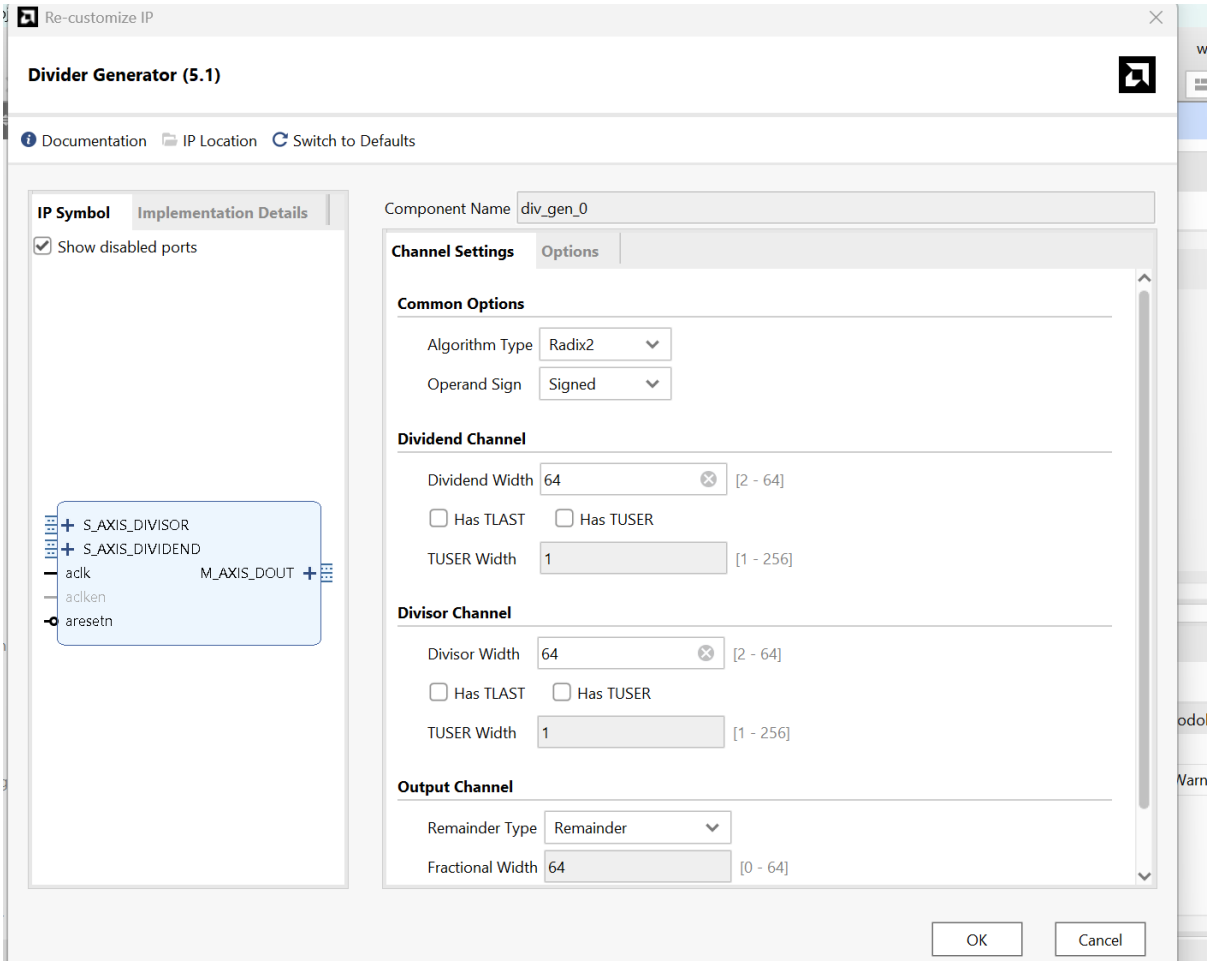

Figure 14: Divider IP signed — Channel Settings fig a

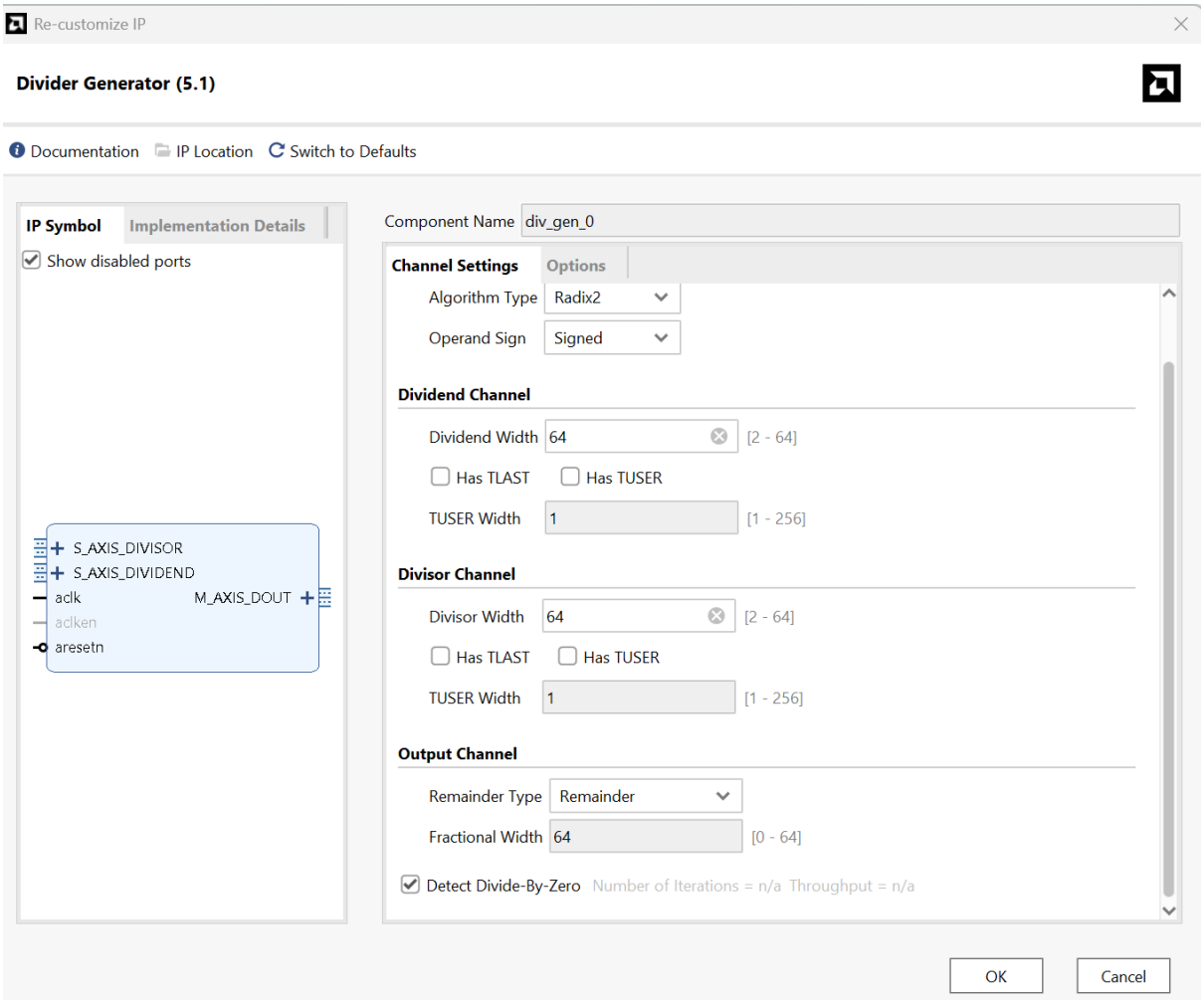

Figure 15: Divider IP signed — Channel Settings fig b

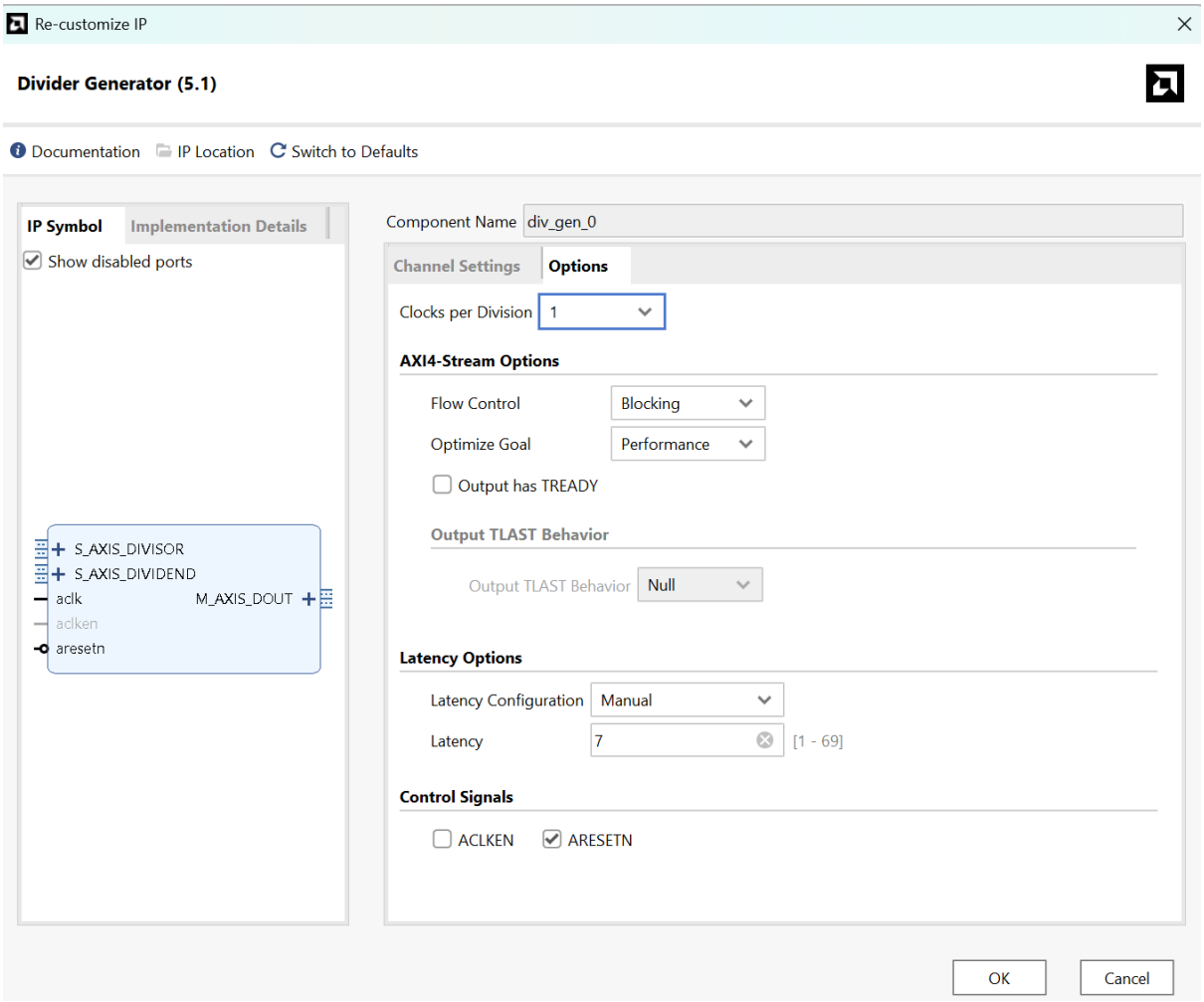

Figure 16: Divider IP signed — Pipeline / Latency Options

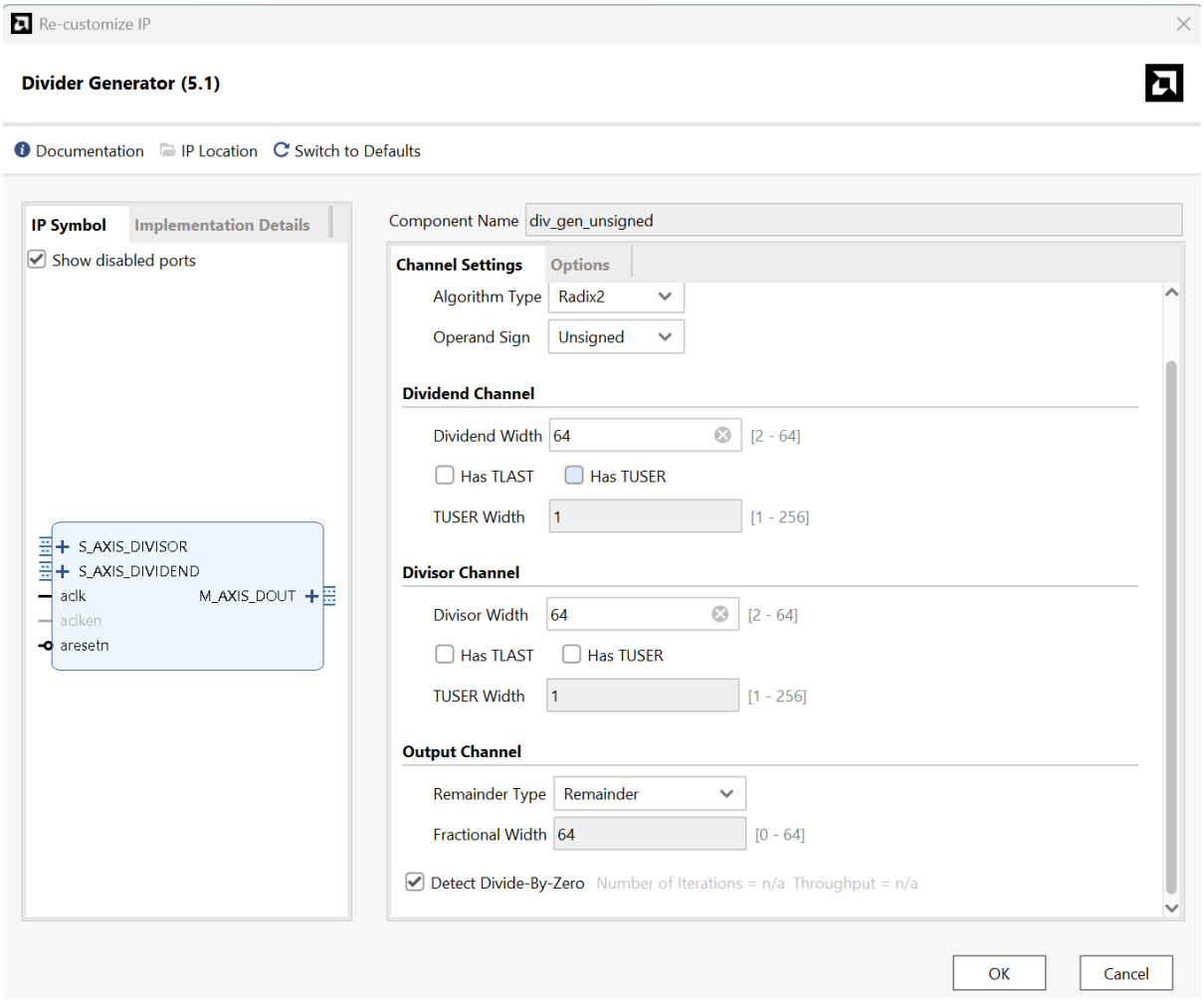

Figure 17: Divider IP unsigned — Channel Settings fig

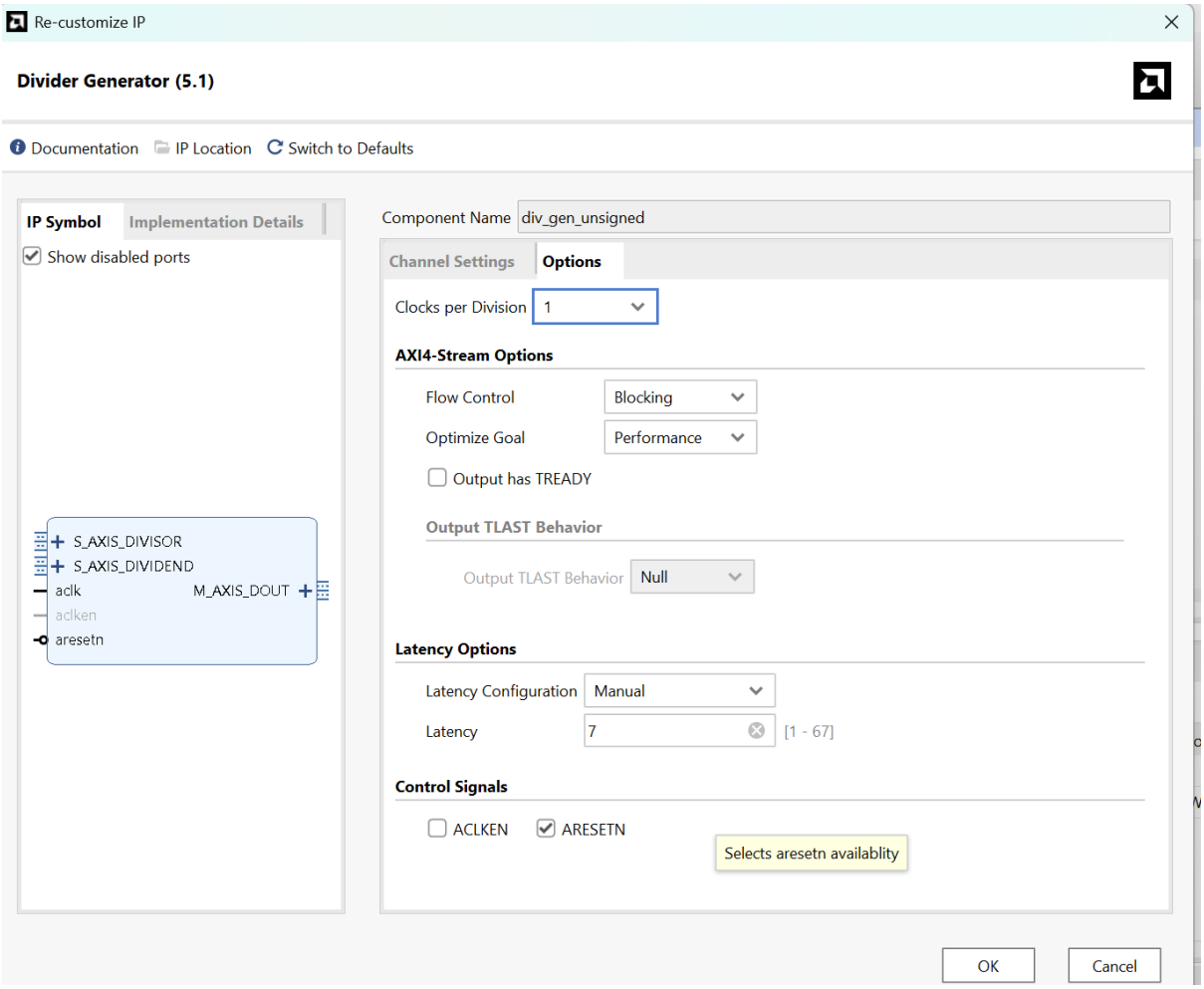

Figure 18: Divider IP unsigned — Pipeline / Latency Options