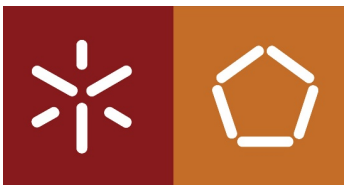

University of Minho
School of Engineering

Processor

GR0040

Group 4

Master's in Industrial Electronics and Computers

André Martins — PG60192
Mariana Martins — PG60211

Project Supervised by
Professor Adriano Tavares

January 2026

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This report is based on the GR0040 System-on-Chip (SoC) implementation provided by the course instructor, which was developed by Jan Gray, from the Gray Research LLC. Serving as a functional reference design for educational purposes and although operating correctly, the existing codebase suffers from significant structural deficiencies. In particular, the design lacks proper separation of concerns, exhibits tight coupling between functional blocks, and provides limited modularity. These issues substantially hinder code readability, verification, debugging, and future extensibility.

1.1 Problem Statement

This project aims to understand the initial GR0040 SoC implementation and to refactor it into a well-structured and modular SoC design while preserving full functional equivalence. The refactoring effort must maintain strict compliance with the original GR0040 architecture and instruction set, ensuring that all existing behaviors remain unchanged. Additionally, any new improvement should be added, as well as new peripherals, such as UART.

This project focuses on analyzing the original implementation, identifying architectural and structural weaknesses, and redesigning the system using a clean, modular architecture. The result should be a robust and maintainable SoC design that supports easier verification, extension, and long-term maintenance, while serving as a solid foundation for future enhancements.

1.2 Motivation

The motivation for this project stems from both educational and practical considerations. From an educational perspective, the original GR0040 implementation provides limited insight into sound hardware design practices due to its poor modular structure and tightly integrated components. Refactoring the system offers an opportunity to apply key digital design principles, such as modularity, abstraction, and separation of datapath and control logic, in a realistic and non-trivial hardware system.

From a practical standpoint, a well-structured SoC design is essential for scalability and verification. As systems grow in complexity, poorly organized designs become increasingly difficult to debug, extend, and validate. By redesigning the GR0040 SoC with clean interfaces and clearly defined modules, this project aims to demonstrate how thoughtful architectural decisions can significantly improve development efficiency and system reliability.

Additionally, the integration of new peripherals, enhanced interrupt handling, and deployment on a real FPGA platform further motivates this work by bridging the gap between theoretical design and real-world hardware implementation. The final refactored system not only validates the correctness of the new architecture but also provides a reusable and extensible platform for future academic or research-oriented developments.

1.3 Requirements

The project requirements are divided into functional and non-functional requirements, addressing both system behavior and quality attributes of the final design.

1.3.1 Functional Requirements

Functional requirements define the expected behavior and capabilities of the system:

- Analyze and verify the functionality of the provided GR0040 and GR0041 processor core implementations.
- Refactor the existing SoC architecture while preserving full functional equivalence with the original design.
- Integrate the existing Timer and Parallel I/O peripherals with enhanced and standardized interrupt support.
- Design and implement a new Universal Asynchronous Receiver-Transmitter (UART) peripheral compatible with the system bus architecture.
- Design and implement an interrupt controller capable of managing interrupts from all peripherals in a scalable and efficient manner.
- Develop and deploy a demonstration program on the Zybo Z7-10 FPGA that validates correct processor operation, peripheral interaction, and interrupt handling.

1.3.2 Non-Functional Requirements

Non-functional requirements specify global quality attributes and design constraints:

- Enforce a clear separation of concerns through function-based and module-based design.
- Refactor the GR0040 processor core to include a strict separation between the datapath and control unit.
- Organize the project in a modular fashion, with each major component implemented in a dedicated Verilog source file.
- Ensure that post-synthesis and post-implementation simulation results match behavioral simulation outcomes.
- Maintain comprehensive documentation describing architectural decisions, module interfaces, and system integration.
- Adhere to Verilog HDL coding best practices to ensure readability, maintainability, and reusability.

1.4 Constraints

Project constraints define the technical, academic, and logistical limitations under which the system must be developed.

1.4.1 Technical Constraints

- Xilinx Vivado Design Suite must be used for synthesis, implementation, and simulation.
- The target hardware platform is the Zybo Z7-10 FPGA development board.
- All system components must be implemented exclusively using Verilog HDL.
- The design must remain fully compliant with the GR0040 instruction set architecture as specified in the reference documentation.
- On-chip block RAM resources must be used for program and data memory, in accordance with the original design specifications.

1.4.2 Non-Technical Constraints

- The project must be completed collaboratively by a team of two members.
- All project deliverables must be completed and submitted by the end of the academic semester.

1.5 Gantt Diagram

Figure 1.1 below illustrates the expected project timeline along with the key milestones to be achieved.

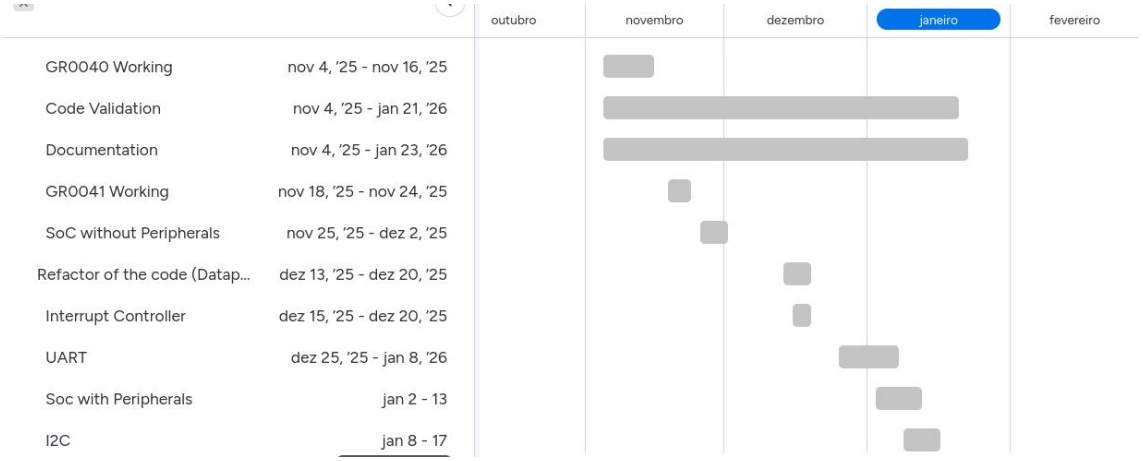

Figure 1.1: Gantt Diagram

1.6 Theoretical Foundation

This section presents the theoretical concepts and design principles that underpin the refactoring and enhancement of the GR0040 System-on-Chip (SoC). The discussion covers processor design fundamentals, hardware description languages, datapath and control unit theory and platform-specific considerations. These concepts form the basis for architectural decisions, implementation strategies, and verification methodologies adopted throughout the project.

1.6.1 Processor Design

A processor is a digital system responsible for executing instructions stored in memory and manipulating data according to a defined instruction set architecture (ISA). At a high level, a processor consists of a datapath, which performs arithmetic and data movement operations,

and a control unit, which orchestrates these operations by generating appropriate control signals.

Modern processor design emphasizes modularity, scalability, and clarity of functionality. In this project, the GR0040 processor core follows a reduced instruction set computing (RISC) paradigm, characterized by simple instructions, uniform instruction formats, and a load/store architecture. These characteristics simplify both the datapath and control logic, making the architecture suitable for educational purposes and hardware refactoring.

The refactoring process relies heavily on classical processor design methodologies, where each functional block (register file, arithmetic logic unit, program counter, memory interface, and control logic) is clearly defined and isolated. This separation enables easier debugging, verification, and future extension of the processor.

1.6.2 Hardware Description Languages (HDL)

Hardware Description Languages are formal languages used to model, simulate, and synthesize digital systems. Unlike software programming languages, HDLs describe hardware behavior and structure concurrently, allowing designers to model parallelism inherent in digital circuits.

Verilog HDL is used throughout this project due to its widespread adoption in industry and full support within the Xilinx Vivado Design Suite. Verilog supports multiple modeling styles:

- **Behavioral modeling**, used for describing functionality without explicitly defining hardware structure.
- **Register-transfer level (RTL) modeling**, which describes how data moves between registers on clock edges and is the primary modeling style used in this project.
- **Structural modeling**, which explicitly instantiates and connects lower-level modules.

Best practices in Verilog coding—such as the use of synchronous logic, non-blocking assignments for sequential circuits, and clearly defined module interfaces—are critical for ensuring synthesizability, timing correctness, and maintainability. These practices guide the refactoring of the original GR0040 implementation.

1.6.3 Processor Datapath Theory

The datapath is the portion of the processor responsible for performing computations and data transfers. It consists of interconnected components such as registers, multiplexers, arithmetic logic units (ALUs), shifters, and memory interfaces.

In a typical RISC datapath, instructions are executed through a sequence of stages, including instruction fetch, decode, execute, memory access, and write-back. Even in non-pipelined implementations, the logical separation of these stages remains conceptually important.

Key datapath components include:

- **Program Counter (PC)**, which holds the address of the current instruction.
- **Instruction Register**, which stores the fetched instruction.
- **Register File**, providing fast access to operands and storage for results.
- **Arithmetic Logic Unit (ALU)**, responsible for arithmetic and logical operations.
- **Multiplexers**, used to select between multiple data sources.

Separating the datapath from the control logic enables a cleaner architecture and simplifies both design verification and future enhancements. This principle is a central requirement of the project.

1.6.4 Control Unit Theory

The control unit governs the operation of the datapath by generating control signals based on the current instruction and processor state. These signals determine actions such as register writes, ALU operation selection, memory access, and program counter updates.

Control units are typically implemented using either:

- **Hardwired control**, where combinational logic directly generates control signals.
- **Microprogrammed control**, where control signals are stored in a control memory.

The GR0040 processor employs a hardwired control unit, which is suitable for simpler RISC architectures. During refactoring, the control logic is explicitly separated from the datapath, improving readability and ensuring that control signal generation can be independently verified.

The control unit also plays a critical role in interrupt handling by coordinating context switching and peripheral interaction.

1.6.5 Target Platform: Zybo Z7-10

The Zybo Z7-10 is an FPGA development board based on the Xilinx Zynq-7000 series, integrating programmable logic with an ARM processing system. In this project, only the programmable logic (PL) is utilized.

The board provides on-chip block RAM, clock management resources, and external interfaces suitable for SoC development. Its compatibility with Vivado enables accurate synthesis, timing analysis, and hardware validation of the refactored design.

1.6.6 RISC Architecture Principles

Reduced Instruction Set Computing (RISC) architectures are designed to simplify processor implementation and improve execution efficiency. Core RISC principles include:

- Simple and fixed-length instructions.
- Load/store architecture, where memory is accessed only through dedicated instructions.
- A large set of general-purpose registers.
- Simplified control logic.

These principles guide the design of the GR0040 processor and strongly influence the refactoring strategy adopted in this project. By adhering to RISC fundamentals, the processor remains efficient, predictable, and easy to extend.

1.6.7 Design and Verification Flow Using Vivado

The Xilinx Vivado Design Suite provides an integrated environment for RTL design, simulation, synthesis, implementation, and bitstream generation. The standard design flow followed in this project includes:

- RTL design and refactoring using Verilog HDL.
- Behavioral simulation to validate functional correctness.

- Synthesis to map RTL to FPGA primitives.
- Implementation (place and route).
- Static timing analysis and hardware validation on the Zybo Z7-10 board.

This flow ensures that the refactored SoC behaves consistently across simulation and hardware execution.

From Verilog Code to Hardware

Although Verilog HDL resembles a programming language, it does not describe a sequence of executed instructions. Instead, Verilog specifies the structure and behavior of digital hardware. Each construct in the code corresponds to a physical hardware element or interconnection once synthesized.

Combinational logic is typically described using continuous assignments or `always @(*)` blocks. These constructs are interpreted by synthesis tools as Boolean logic, which is later mapped to lookup tables (LUTs) in the FPGA fabric. For example, arithmetic operations, multiplexers, and logic comparisons are implemented as networks of LUTs.

Sequential logic is described using clocked `always` blocks, commonly of the form `always @(posedge clk)`. When such a block assigns values to registers using non-blocking assignments, the synthesis tool infers flip-flops. Each variable assigned in this context corresponds to a physical storage element triggered by the specified clock edge.

As a result, the structure of the Verilog code directly determines the resulting hardware. Poorly structured code can lead to unintended latches, excessive logic, or timing issues, whereas well-structured RTL code results in predictable and efficient hardware implementations.

Inference of Flip-Flops and Registers

Flip-flops are the fundamental storage elements in synchronous digital systems. In Verilog, flip-flops are inferred rather than explicitly instantiated. A flip-flop is inferred when the following conditions are met:

- The signal is assigned within a clocked `always` block.
- The block is sensitive to a clock edge (e.g., `posedge clk`).
- The signal is assigned in all possible execution paths of the block.

For example, a register in the processor datapath—such as the program counter or a general-purpose register—is inferred as a bank of flip-flops, with each bit mapped to a physical flip-flop on the FPGA. Reset logic, if present, is implemented using either synchronous or asynchronous reset ports of the flip-flops, depending on the coding style.

This inference mechanism allows designers to describe high-level behavior while relying on synthesis tools to generate the appropriate low-level hardware structures.

Synthesis Process

Synthesis is the process of translating Verilog RTL code into a gate-level representation tailored to the target FPGA technology. During synthesis, the Vivado tool performs several key steps:

- **RTL elaboration**: The Verilog code is parsed, and all module hierarchies, parameters, and signal connections are resolved.
- **Logic inference**: Combinational logic, flip-flops, multiplexers, memories, and finite state machines are identified based on coding patterns.

- **Technology mapping**: The inferred logic is mapped to FPGA-specific primitives such as LUTs, flip-flops, block RAMs, and DSP slices.
- **Optimization**: Redundant logic is removed, Boolean expressions are simplified, and logic is reorganized to improve area and timing.

The output of synthesis is a technology-specific netlist that describes how the design will be implemented using the resources available on the Zybo Z7-10 FPGA.

Implementation: Place and Route

Following synthesis, the design undergoes implementation, which consists primarily of placement and routing:

- **Placement**: Each synthesized logic element (LUTs, flip-flops, block RAMs) is assigned to a specific physical location on the FPGA fabric.
- **Routing**: Electrical connections between placed elements are established using the FPGA's programmable interconnect network.

During this phase, Vivado considers timing constraints, resource availability, and routing congestion. The tool attempts to minimize critical path delays while ensuring all connections can be successfully routed.

Clock networks receive special treatment to minimize skew and jitter. Dedicated clock routing resources are used to distribute the clock signal uniformly across the design.

Bitstream Generation and Hardware Execution

Once timing closure is achieved, Vivado generates a configuration bitstream. This bitstream programs the FPGA by defining the state of its programmable logic and routing resources.

After configuration, the FPGA behaves as a custom hardware implementation of the Verilog design. The processor core, peripherals, memories, and interconnect operate concurrently, executing instructions and responding to events in real time.

This transformation—from high-level Verilog code to physical hardware—highlights the importance of correct RTL design, as errors in the code directly translate into faulty hardware behavior.

Chapter 2

Analysis

This chapter presents the original GR0040 RISC processor architecture designed by Jan Gray, which serves as the foundation for our implementation. The architecture is specifically optimized for FPGA implementation, making efficient use of Xilinx Virtex-II resources such as block RAMs, LUTs, and tri-state buffers.

The GR0040 is a 16-bit non-pipelined RISC processor featuring a compact instruction set, Harvard architecture with separate instruction and data paths, and a unique interrupt handling mechanism implemented outside the core. The complete design details, including synthesizable Verilog code and implementation rationale, can be found in the original paper referenced in the bibliography.

The following sections present the key architectural components through a series of progressive design stages, from the basic instruction set to the complete system-on-chip with peripherals.

2.1 Instruction Set Architecture

The GR0040 instruction set is organized into five formats (rr, ri, rri, i12, and br), supporting 22 operations plus 16 branch variants. This design minimizes instruction decode complexity while maintaining sufficient expressiveness for integer C code execution.

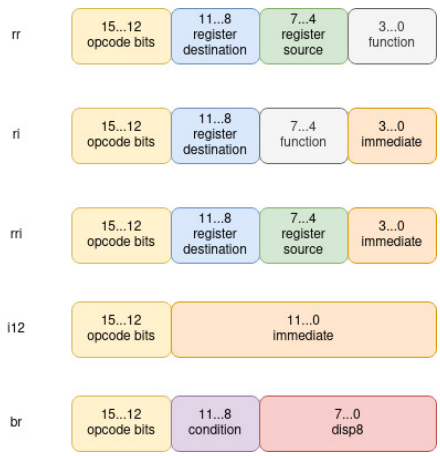

Figure 2.1: GR0040 instruction formats: rr (register-register), ri (register-immediate), rri (register-register-immediate), i12 (12-bit immediate), and br (branch with condition and displacement)

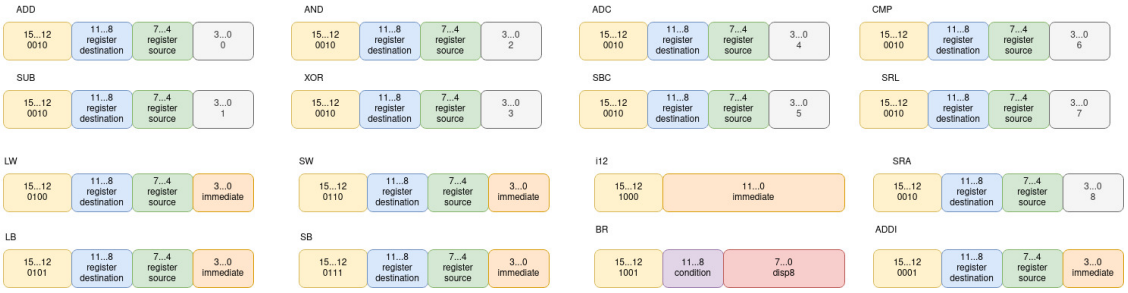

Figure 2.2: Complete GR0040 instruction set showing opcode assignments, formats, assembly syntax, and semantics. Notable features include interlocked instructions (imm, adc*, sbc*, *cmp*) and efficient immediate constant handling

2.2 Memory Organization

The system employs a Harvard architecture with 1KB of on-chip block RAM shared between program and data storage. The memory map reserves the lower 1KB for on-chip RAM and dedicates addresses 0x8000-0xFFFF to memory-mapped peripheral I/O.

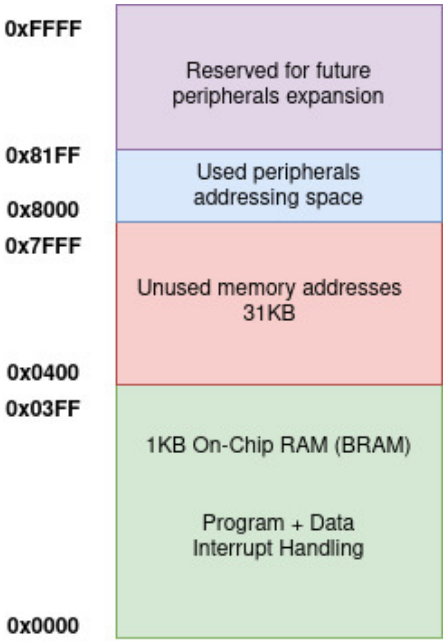

Figure 2.3: Memory address space distribution: 1KB on-chip BRAM (0x0000-0x03FF), unused address space (0x0400-0x7FFF), and memory-mapped peripherals (0x8000-0xFFFF)

2.3 Core Processor Module (GR0040)

The GR0040 core implements the basic processor functionality with separate instruction fetch and load/store data ports. The core assumes instruction availability from zero-latency block RAM, enabling single-cycle instruction execution for most operations.

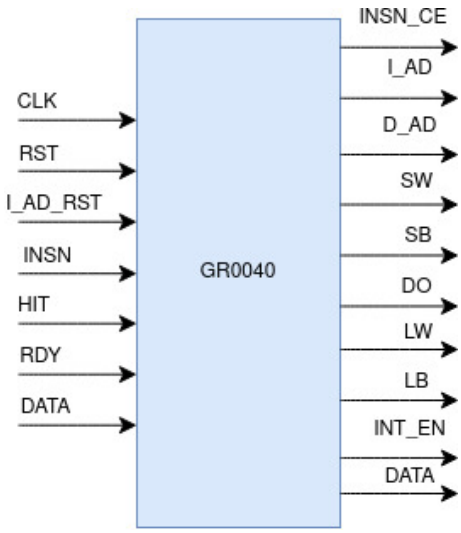

Figure 2.4: GR0040 core interface signals: instruction port (insn_ce, i_ad, insn, hit), data port (d_ad, rdy, sw, sb, do, lw, lb, data), and interrupt support (int_en)

The GR0040 core is composed of several key functional modules that work together to execute instructions:

- **Instruction Decoder**: Splits the current instruction into constituent fields (opcode, register addresses, function codes, immediates) and generates control signals for the datapath.
- **Register File**: A 16×16-bit dual-port register file implemented using distributed RAM (LUTs configured as RAM). It provides asynchronous read access to two registers (rd and rs) simultaneously and synchronous write-back of results.
- **Immediate Operand Formation**: Handles sign/zero extension of 4-bit immediate fields and supports the `imm` prefix instruction for constructing 16-bit immediates by concatenating a 12-bit prefix with the 4-bit immediate field.
- **Operand Selection**: Multiplexers that route either register values or immediate constants to the ALU inputs, with special handling for ri-format instructions.
- **Arithmetic Logic Unit (ALU)**: Comprises three parallel functional units:
  - *Adder/Subtractor*: A 16-bit ripple-carry adder/subtractor with carry-in/carry-out support for multi-precision arithmetic (adc/sbc instructions)
  - *Logic Unit*: Performs AND and XOR operations
  - *Shift Right Unit*: Implements logical and arithmetic right shift by one bit position
- **Condition Code Logic**: Generates and maintains hidden condition flags (zero, negative, carry, overflow) for conditional branch evaluation. The overflow flag uses the clever identity: $v = c[W] \oplus sum[N] \oplus a[N] \oplus b[N]$ to avoid needing both carry-out bits.
- **Result Multiplexer**: Selects the final result from among ALU outputs, PC (for JAL), or loaded data, using tri-state buffers on FPGA long-lines to implement an efficient multi-input mux without consuming LUTs.
- **Branch Evaluation**: Decodes the branch condition field and compares it against stored condition codes to determine if branches are taken.

- **Address/PC Unit**: Computes the next program counter value, handling sequential execution, branch targets (PC + 2×sign-extended displacement), and jump targets (effective address from adder).

This modular organization enables a compact implementation consuming only about 200 logic cells in a Spartan-II FPGA, while achieving clock speeds of 50 MHz with careful optimization.

2.4 Interrupt Controller Module (GR0041)

The GR0041 module wraps the GR0040 core with external interrupt handling logic. This modular approach inserts a "call intr" instruction (JAL r0,2(r0)) into the instruction stream upon interrupt request, leveraging the block RAM reset functionality to force instruction 0x0000 and then OR it with 0x0002.

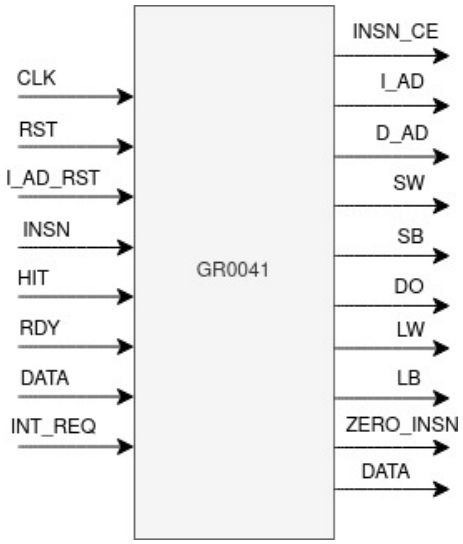

Figure 2.5: GR0041 interrupt wrapper interface, adding int_req input and zero_insn output to force instruction RAM output during interrupt insertion

The interface between GR0040 and GR0041 consists of two critical signals that enable interrupt handling without modifying the core processor:

- **int_en** (output from GR0040): Indicates when the processor is in an interruptible state. This signal is deasserted during interlocked instruction sequences (imm, adc*/sbc*, *cmp*) to prevent interrupts from breaking multi-instruction atomic operations.
- **int_req** (input to GR0041): The interrupt request signal from peripherals. GR0041 monitors this along with int_en to determine when to insert the interrupt instruction.
- **zero_insn** (output from GR0041): When asserted, forces the block RAM to output 0x0000, which is then modified to 0x0002 (the JAL r0,2(r0) instruction) before being presented to the GR0040 core.

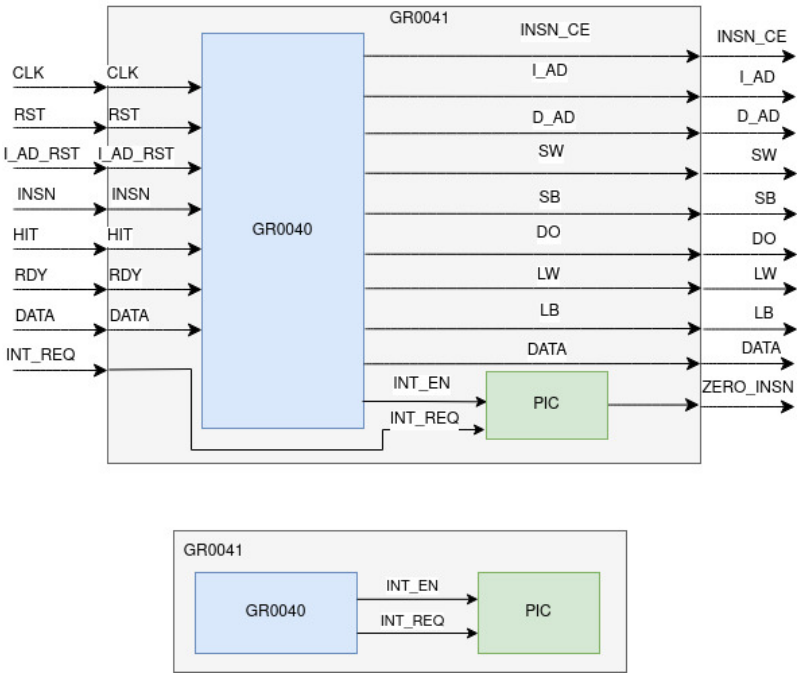

Figure 2.6: Hierarchical relationship between GR0040 core and GR0041 interrupt wrapper, showing how the PIC (Programmable Interrupt Controller) mediates interrupt requests

The interrupt mechanism operates through rising-edge detection on int_req, maintaining an internal int_pend flag until the interrupt can be serviced. Figure 2.7 illustrates the complete interrupt flow, showing how the interrupt instruction is inserted and how the interrupt service routine (ISR) preserves the return address temporarily in r0 before restoring it upon return.

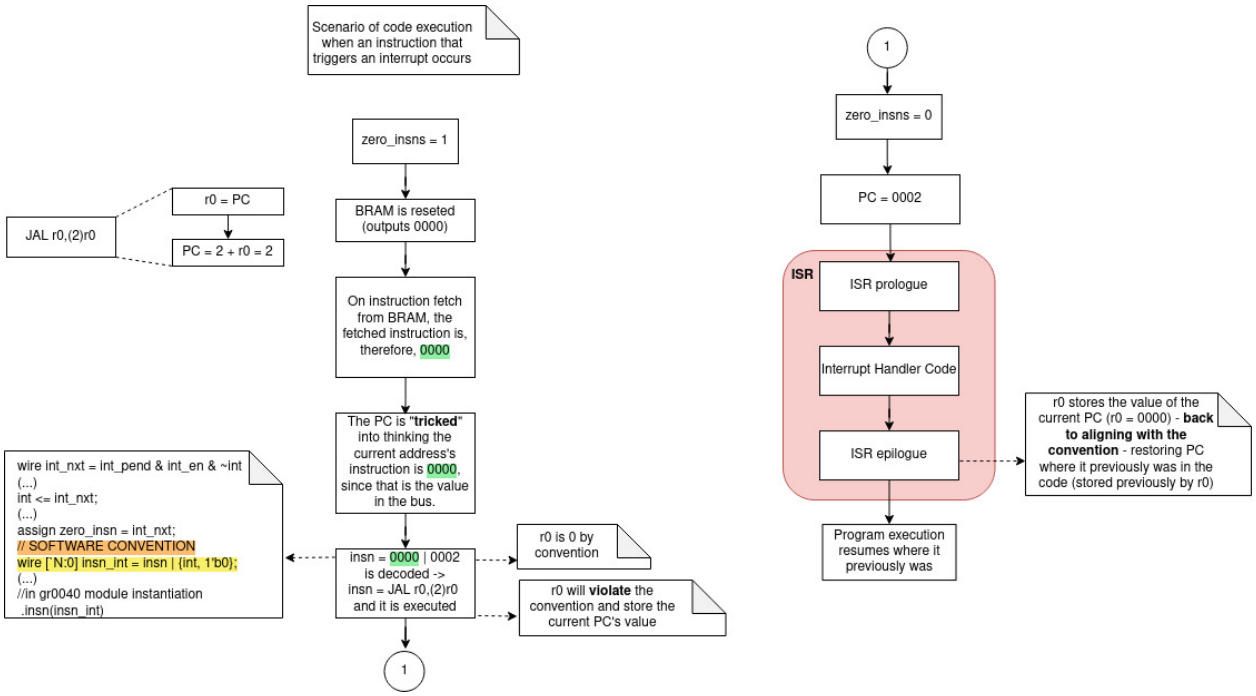

Figure 2.7: Interrupt handling mechanism: (1) interrupt detection and instruction insertion flow, showing how zero_insn forces BRAM output to 0x0000, which is then modified to 0x0002 (JAL r0,2(r0)) to call the interrupt handler; (2) interrupt service routine structure with prologue, handler code, and epilogue that preserves the interrupted instruction using r0 as temporary storage

This clever design exploits the fact that JAL has opcode 0, so forcing the block RAM output to 0x0000 and ORing with 0x0002 produces exactly the needed interrupt call instruction, at minimal hardware cost.

2.5 Complete System-on-Chip with Peripherals

The final SoC integrates the GR0041 processor core with dual-ported block RAM (configured as separate high and low byte RAMs for byte-addressable access), an abstract control bus architecture (ctrl, sel, per_rdy), and memory-mapped peripherals including a timer/counter and 8-bit parallel I/O port.

2.5.1 Block RAM Implementation

The system uses two Xilinx RAMB4_S8_S8 block RAM primitives (ramh and raml) configured as dual 512×8-bit memories, providing a total of 1KB of storage. Each block RAM has two independent ports:

- **Port A**: Dedicated instruction fetch port (read-only), delivering 16-bit instructions by combining one byte from ramh[15:8] and one byte from raml[7:0]
- **Port B**: Data load/store port (read-write), supporting both word and byte accesses with appropriate byte-write-enable logic

This dual-port configuration enables simultaneous instruction fetch and data access without contention, essential for maintaining single-cycle throughput. For byte stores, the design routes the data byte to either ramh or raml based on the address LSB, since the architecture is big-endian.

2.5.2 On-Chip Peripheral Bus Architecture

The system employs an abstract control bus architecture designed for simplicity and extensibility. Rather than exposing individual control signals, the design encapsulates all peripheral control information into three buses:

- **ctrl[31:0]**: Abstract control bus carrying encoded clock, reset, I/O address, byte output enables (oe[3:0]), and byte write enables (we[3:0])
- **sel[7:0]**: Peripheral select vector, with each bit selecting one of up to 8 peripherals based on address decoding
- **per_rdy[7:0]**: Ready signals from each peripheral, OR-reduced to form the global io_rdy signal
- **data[15:0]**: Shared tri-state data bus for both processor results and peripheral I/O

The **ctrl_enc** module encodes processor control signals and I/O addresses into the abstract ctrl bus, while each peripheral instantiates a **ctrl_dec** module to decode the signals it needs. This abstraction layer provides several benefits:

- Core users only need to connect four buses to add a peripheral
- The bus protocol can evolve without invalidating existing peripheral designs
- Address decoding is implicit through sel[] assignment
- Typical peripheral bus interface overhead is only 1-2 CLBs plus a column of tri-state buffers

2.5.3 Integrated Peripherals

As shown in Figure 2.8, the SoC includes two memory-mapped peripherals:

- **Timer/Counter**: A 16-bit configurable timer/counter peripheral assigned to sel[0] (addresses 0x8000-0x80FF). It operates in timer mode (counting clock cycles when enabled) or counter mode (counting rising edges on an external input). The timer includes a programmable initial count value, overflow detection, and interrupt generation capability. In the default configuration, it counts from 0xFFC0 to overflow, generating an interrupt every 64 clock cycles.
- **Parallel I/O Port (PARIO)**: An 8-bit bidirectional parallel I/O port assigned to sel[1] (addresses 0x8100-0x81FF). It provides simple memory-mapped access to external parallel inputs and outputs, suitable for interfacing with LEDs, switches, or other slow peripherals. The peripheral asserts rdy immediately upon selection, requiring no wait states.

Both peripherals connect to the abstract control bus and tri-state data bus, demonstrating the ease of integrating additional functionality into the SoC framework.

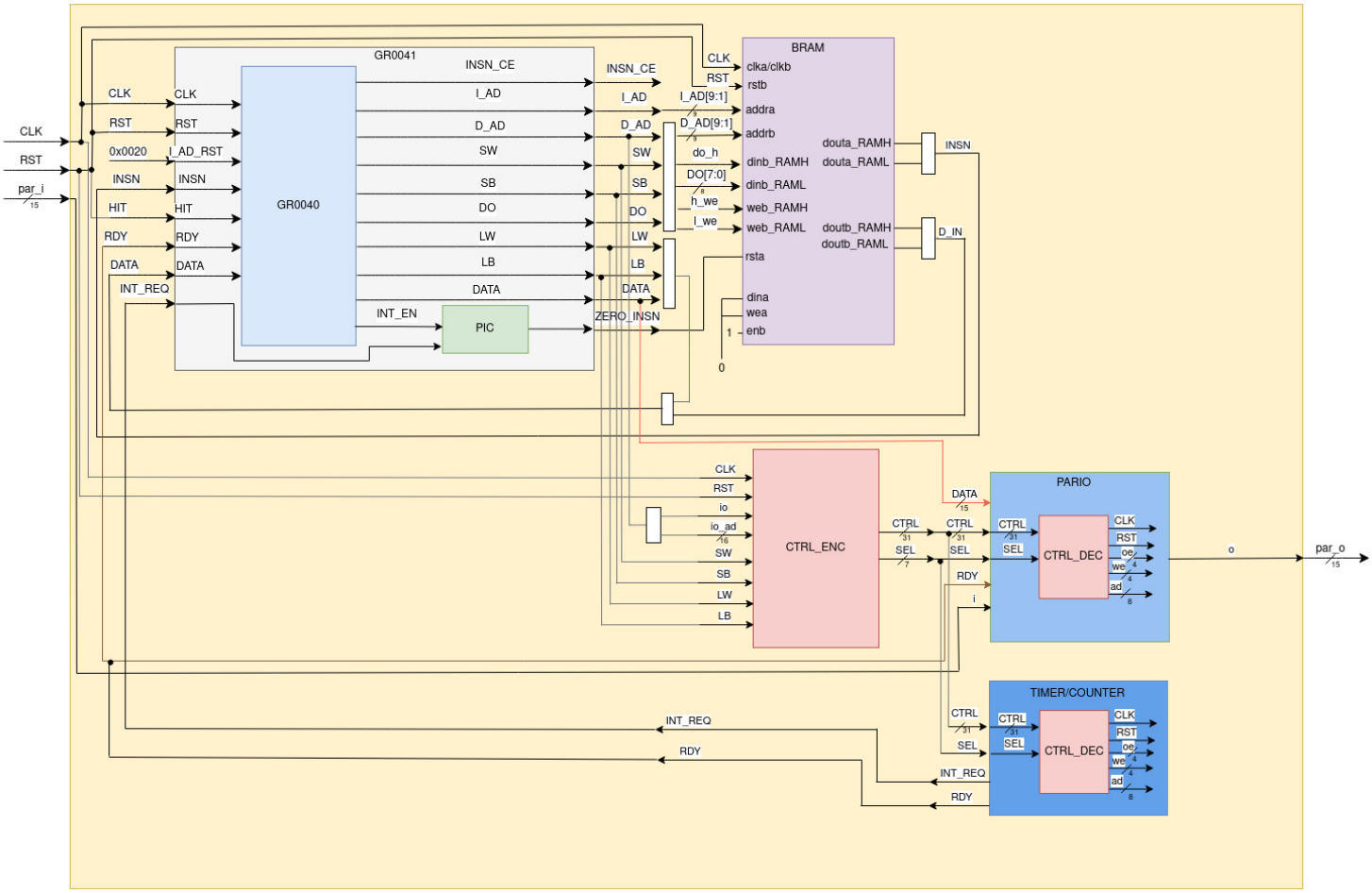

Figure 2.8: Complete system-on-chip architecture showing GR0041 processor with interrupt controller, dual 512×8 block RAMs (ramh, raml) for 1KB program/data storage, ctrl_enc/ctrl_dec bus interface modules, and two peripherals (PARIO and TIMER/-COUNTER) connected via the abstract control bus and tri-state data bus

Chapter 3

Design- Refactoring

3.1 Assembler

In order to simplify instruction decoding from Assembly to its equivalent hexadecimal value, an Assembler was developed in C. For a given Assembly code with basic sections, such as `isr:`, `reset:` and `main:`, the Assembler decodes the instructions and their operands and transforms them into their hexadecimal equivalent (Figure 3.2). A `.asm` file is edited and the opcodes are generated in the correct and desired order, through a `makefile` (Figure 3.1), which simply calls the C compiler and the necessary files and verification flags.

Additionally, it separates the instructions for the High and Low memory `.coe` files used.

```
andre@macaco:~/Documents/Assembler$ make
gcc -Wall -o program main.c assembler.c
./program instructions.asm hexa.txt
Program Execution Terminated. Opening Output File
xdg-open hexa.txt
```

Figure 3.1: `make` result

Open | instructions.asm ~/Documents/Assem... | Save

```
1 isr:
2 // Interrupt handler
3 JAL R0, R0, #0
4 ADDI R15, R15, #-1
5 SW R1, R15, #0
6 BR #-3
7
8 reset:
9 // Reset vector at 0x0020
10 IMM #128
11 ADDI R15, R0, #2
12
13 start:
14 // Main program
15 ADDI R1, R0, #10
16 ADDI R2, R0, #15
17 ADD R3, R1
18 CMP R1, R2
19 BEQ #5
```

(a) Assembly Instructions

Open | hexa.txt ~/Downloads

```
1 0000
2 1FFF
3 61F0
4 90FD
5 FFFF
6 FFFF
7 FFFF
8 FFFF
9 FFFF
10 FFFF
11 FFFF
12 FFFF
13 FFFF
14 FFFF
15 FFFF
16 FFFF
17 8080
18 1F02
19 110A
20 120F
21 2310
22 2126
23 9205
```

(b) Hexadecimal Equivalent

Figure 3.2: Comparison between Assembly Instructions and their Hexadecimal Equivalent

3.2 Signal Naming Standardization

During the design phase of our FPGA-based processor, one of the first tasks was the refactoring of the code, focusing on signal naming standardization. Refactoring restructures the code to improve readability, maintainability, and scalability without changing its functionality. Standardized signal names ensure the design is consistent, intuitive, and easier to debug or extend.

Initial Signal Naming

The initial code used short or non-descriptive names, which can be ambiguous in larger designs:

```
reg [`AN:0] pc;          // program counter
output [`AN:0] i_ad;     // next instruction address
output [`AN:0] d_ad;     // data address for load/store
output [`N:0] do;        // data to store in RAM
wire [`N:0] dreg, sreg; //destination and source register
```

These names do not clearly indicate the signal's role or width. For example, `pc` could refer to the current or next program counter, and `do` is too generic for RAM write data.

Refactored Signal Naming

After refactoring, the signals follow a consistent convention:

- **Ports:** Prefixed with `i_` for inputs and `o_` for outputs.
- **Internal signals:** Prefixed with `_` to indicate internal use within a module.
- **Descriptive names:** Each signal clearly reflects its function.
- **Parameter-based widths:** Widths defined using global macros (e.g., `ADDR_MSB`, `DATA_MSB`).

```
reg [`ADDR_MSB:0] _IR;              // current instruction register
wire [`ADDR_MSB:0] _PC;             // next instruction address
wire [`ADDR_MSB:0] _data_address;   // data address for load/store
wire [`DATA_MSB:0] _data_to_store; // data to write to RAM
wire [`DATA_MSB:0] _destinationRegister;
wire [`DATA_MSB:0] _2ndOp;

input i_clk;                          // system clock
output [`DATA_MSB:0] o_insn;          // current instruction output
```

This refactoring also aligns with microcontroller naming conventions, such as those used in the 8051, particularly in UART modules in section **??**.

These conventions improve code clarity, reduce potential errors, and facilitate integration with other modules. Using these conventions consistently across the processor ensures readability and simplifies verification and testing.

3.3 Instruction Set Architecture

The modifications to the original Instruction Set Architecture (ISA) were minimal and focused primarily on enhancing the RR (register-register) format instructions and adding new instructions. As shown in Figure **??**, the original ISA specification defined five main instruction formats: `rr`, `ri`, `rri`, `il2`, and `br`, each with specific field allocations for opcode, register destinations, register sources, and immediate values.

The primary enhancements involved two key additions: expanding the functionality within the RR format to include additional shift register operations, and introducing the RETI (Return from Interrupt) instruction for interrupt handling support.

Shift Register Operations

Figure 3.3 illustrates the extended instruction encodings for shift operations, which now support four distinct shift operations: Shift Right Logic (SRL), Shift Left Arithmetic (SLA), Shift Right Arithmetic (SRA), and Shift Left Logic (SLL). These operations maintain the same field structure as the original RR format, utilizing bits 15-12 for the opcode (0010), bits 11-8 for the register destination, bits 7-4 for the register source, and bits 3-0 to encode the specific shift operation type.

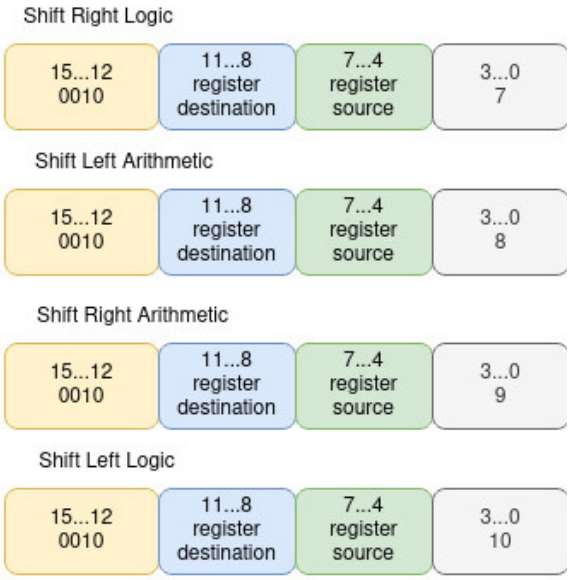

Figure 3.3: Shift Operation Refactor

The decoding logic for these shift operations, depicted in Figure 3.4, employs a compact 2-bit selection scheme to differentiate between the four shift types. The color-coded matrix shows how different bit combinations map to logic operations (green), arithmetic operations (purple), shift left operations (blue), and shift right operations (yellow). This encoding allows for efficient hardware implementation while maintaining clarity in the instruction semantics.

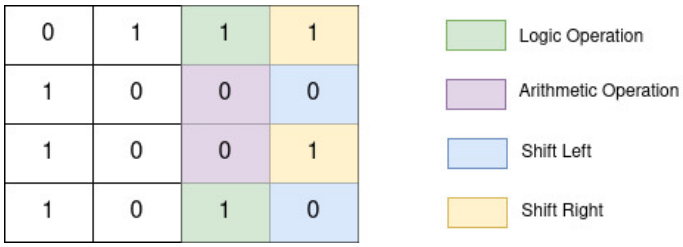

Figure 3.4: L0gic behing shift operation refactoring

Return from Interrupt (RETI) Instruction

A significant architectural enhancement was the introduction of the RETI instruction, as illustrated in Figure 3.5. Unlike traditional branch instructions that require explicit specification of jump offsets or target addresses, RETI provides a dedicated mechanism

for returning from interrupt service routines. This instruction eliminates the need for manually calculating and encoding branch displacements when exiting interrupt handlers, simplifying interrupt management and reducing potential errors in interrupt-driven code.

Figure 3.5: RETI instruction

The RETI instruction operates by automatically restoring the program counter to the address that was stored when the interrupt was triggered. This approach streamlines the interrupt handling process and ensures proper return flow control without requiring programmer intervention to manage return addresses explicitly.

This architectural extension demonstrates how the original ISA can be enhanced with minimal modifications to the instruction format while adding significant computational capability through comprehensive shift operations and improved interrupt handling support. The interrupt handling mechanism will be discussed in detail in the following sections.

3.4 Register File

The original code was developed for the Spartan-II, a member of the Virtex family of FPGAs, and relied on memory resources and IP cores that are no longer optimal for modern platforms. Since the target platform for this project is the ZYBO development board, which is based on a Xilinx Zynq device, a redesign of the register file implementation was required.

In the refactored design, the register file was implemented using the **Distributed Memory Generator** IP core. This choice was motivated by the relatively small size of the register file and the need for fast, deterministic access. Distributed memory maps directly onto lookup tables (LUTs), allowing for low-latency read and write operations, which is particularly advantageous for register files accessed every clock cycle.

Signal	Direction	Description
a	Input	Register write address
d	Input	Data to be written to the register
dpra	Input	Dual-port read address
clk	Input	System clock
we	Input	Register file write enable
spo	Output	Single-port read output
dpo	Output	Dual-port read output

Table 3.1: Signals of the `regfile16x16` module

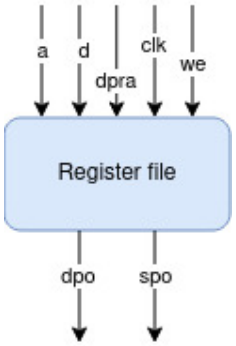

Figure 3.6: Inputs and Outputs of Register file

Additionally, using distributed memory provides greater flexibility in port configuration compared to block RAM, enabling efficient implementation of multiple read and write ports without incurring the complexity or resource overhead associated with BRAM-based solutions. This approach also aligns well with the architectural requirements of the processor, where compact storage and high-speed access are prioritized over large capacity.

3.5 Memory Architecture Refinement

In the refactoring process, a significant architectural decision was made to separate the original unified Block RAM (BRAM) structure into distinct Read-Only Memory (ROM) and Random Access Memory (RAM) modules. This separation provides better organization, enhanced security, and improved system performance compared to the monolithic memory approach used in the original implementation.

The original architecture, as illustrated in Figure 2.3, employed a single 1KB on-chip BRAM that stored both program instructions and data in a shared address space. While this approach is simple and resource-efficient, it presents several limitations:

- **Security concerns**: Executable code sharing the same memory space as data creates potential vulnerabilities, as errant writes could corrupt program instructions.
- **Limited scalability**: As the system grows, managing a single unified memory becomes increasingly complex.
- **Inflexibility**: Different access patterns for instructions (read-only) and data (read-write) cannot be optimized independently.

The refactored architecture addresses the limitations of the original design by adopting a Harvard-style memory organization, in which instruction and data memories are physically separated. In this implementation, program instructions are stored in a dedicated ROM, while data is accessed through a separate RAM, as illustrated in Figure 3.7.

In the original unified memory approach, instructions and data shared the same address space, increasing the risk of unintended memory corruption and making it more difficult to reason about memory accesses during execution. By contrast, the Harvard architecture inherently protects program code, since instructions stored in ROM cannot be altered during runtime. This guarantees program integrity and eliminates an entire class of errors related to accidental overwriting of instruction memory.

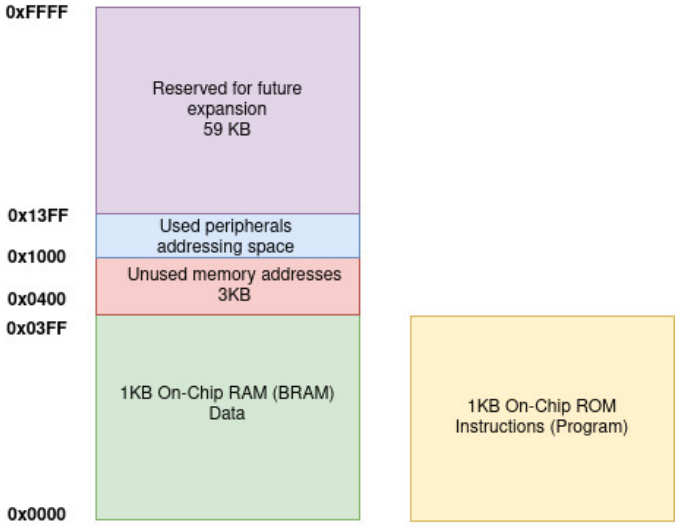

Figure 3.7: Refactored memory architecture with separate ROM and RA

From a performance and design perspective, separating ROM and RAM allows each memory to be optimized for its specific access pattern. Instruction memory can be tailored for sequential fetch operations, while data memory can be optimized for frequent read and write accesses. This separation simplifies timing analysis and improves determinism, which is particularly important in FPGA-based designs.

Furthermore, the physical distinction between instruction and data memories enforces a clearer conceptual model of the processor. During development and debugging, it becomes easier to identify whether an issue originates from instruction execution or data manipulation, significantly reducing debugging complexity. This clarity also improves code readability and maintainability, especially as the system scales or additional peripherals are integrated.

Both the ROM and RAM modules were implemented using Xilinx Block Memory Generator IP cores, each configured for 1KB capacity with single clock cycle latency. This zero-cycle latency is critical for maintaining high performance, as it allows instruction fetch and data access to complete within a single clock cycle, eliminating the need for wait states in most operations.

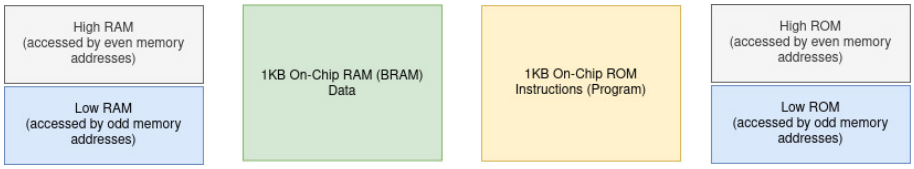

Figure 3.8: Specification of on-chip RAM and ROM

Following the design principles established in the original implementation, we maintained the high/low memory organization for both ROM and RAM. This decision was driven by practical considerations: the byte-addressable nature of the memory system requires separate control of individual bytes, and the high/low split simplifies address decoding logic.

This organization maintains compatibility with byte-wide operations (LB, SB) while efficiently supporting word-wide accesses (LW, SW) through simultaneous access to both high and low memory banks.

Peripheral Address Space Reorganization

The peripheral addressing scheme was also refined in the refactored design. Instead of using the original address range of `0x8000` to `0xFFFF`, the peripheral space now occupies addresses `0x1000` to `0x13FF`. This decision was motivated by several factors:

- **Reduced address gap**: Positioning peripherals immediately after the BRAM/ROM space minimizes unused address ranges, improving memory map efficiency.
- **Current requirements**: With only four peripherals currently implemented, the 768-byte range (`0x1000-0x13FF`) provides ample addressing space.
- **Future expansion**: This organization reserves the upper address space (`0x13FF-0xFFFF`) for future peripheral additions while maintaining a clean, organized memory map.

3.6 Datapath

The datapath represents the execution engine of the processor, containing all the functional units and data storage elements necessary to perform instruction operations. As shown in Figure 3.9, the datapath receives control signals from the Control Unit and executes the actual computational work, including arithmetic operations, logical operations, data movement, and memory access.

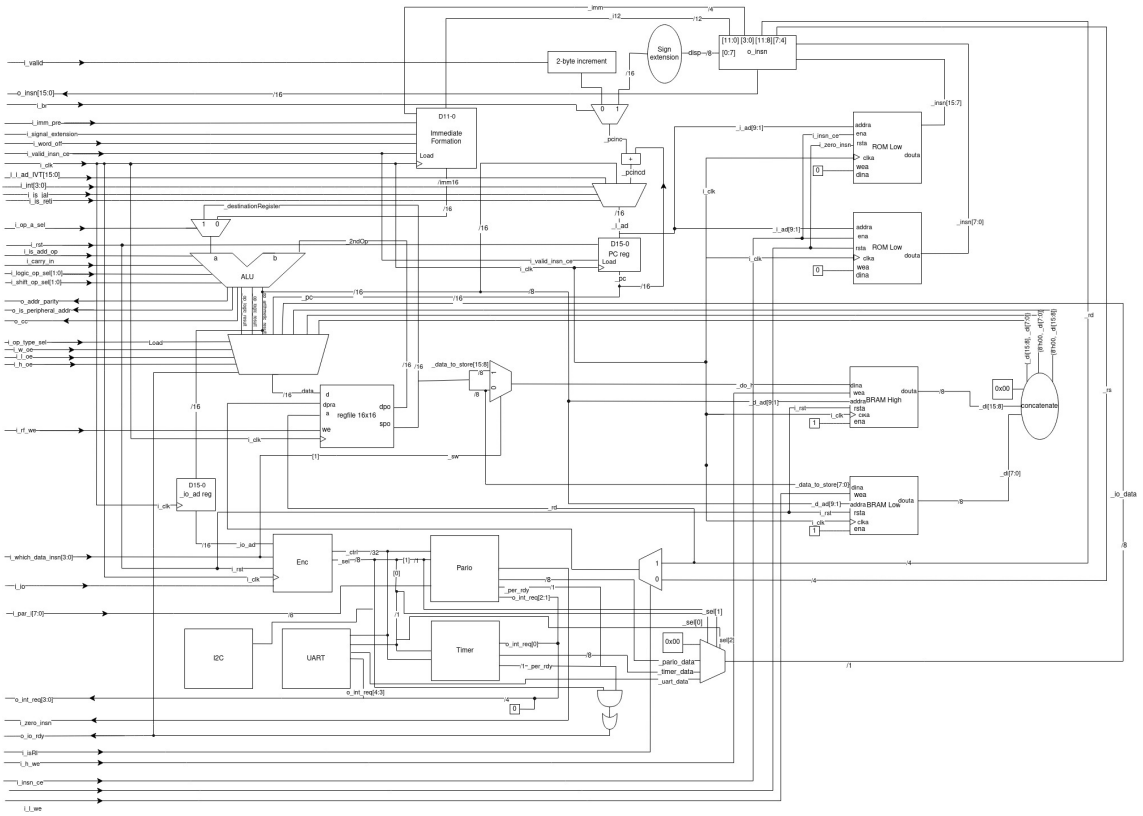

Figure 3.9: Complete datapath architecture showing all functional units and data flow paths

3.6.1 Datapath Overview

The datapath architecture follows a classical RISC design philosophy with distinct functional blocks connected through multiplexers and buses. The main components visible in the architecture include:

- **Register File (regfile 16x16)**: A 16-register by 16-bit dual-port register file that provides fast access to operands and storage for computation results
- **Arithmetic Logic Unit (ALU)**: Performs arithmetic operations (addition, subtraction), logical operations (AND, XOR) and shift operations (SLL, SRL, SLA, SRA) on operands
- **Immediate Formation Unit**: Constructs immediate operands from instruction fields, supporting sign extension and word offset modes
- **Program Counter (PC) Logic**: Manages instruction sequencing, including sequential execution, branches, and jumps
- **Memory Interface**: Connects to ROM (for instructions) and BRAM (for data), handling both word and byte accesses through separate high and low memory banks
- **Peripheral Interface**: Provides connectivity to on-chip peripherals including UART, Timer, and Parallel I/O (Pario) modules
- **Interrupt Controller (Enc)**: Manages interrupt requests from peripherals and coordinates interrupt handling with the Control Unit

The datapath operates synchronously with the system clock, with data flowing from left to right through the various functional units. Control signals from the Control Unit configure multiplexers to route data appropriately and enable specific operations in each functional unit based on the current instruction being executed.

3.6.2 Internal Modules

The datapath is composed of several specialized modules, each responsible for specific aspects of instruction execution. This modular design promotes clarity, reusability, and ease of verification.

Register File

The register file module (regfile 16x16) implements a dual-port register bank with 16 general-purpose 16-bit registers (r0-r15). It features:

- **Dual-port architecture**: Allows simultaneous read of two operands (destination and source registers) in a single cycle
- **Synchronous write**: Results are written back to the destination register on the rising edge of the clock when the write enable signal is asserted
- **Asynchronous read**: Operands are available combinatorially after address inputs stabilize, minimizing latency
- **r0 convention**: By software convention, r0 is maintained as zero, though it can be temporarily used during interrupt handling

Arithmetic Logic Unit (ALU)

The ALU is the computational heart of the datapath, performing all arithmetic and logical operations. It consists of three parallel functional units:

- **Adder/Subtractor**: Implements 16-bit addition and subtraction with carry-in and carry-out support for multi-precision arithmetic. The unit also generates condition codes (Zero, Negative, Carry, Overflow) used by conditional branch instructions.

- **Logic Unit**: Performs bitwise AND and XOR operations, selected by the logic operation select signal from the Control Unit.
- **Shift Unit**: Executes right shift operations including logical right shift (SRL) and arithmetic right shift (SRA), as described in the ISA modifications section.

The ALU outputs are multiplexed based on the operation type, with the selected result driven onto the internal data bus for writeback to the register file or memory.

Immediate Formation Unit

This module constructs immediate operands from instruction fields according to the instruction format:

- **Sign extension**: Extends 4-bit immediate values to 16 bits, preserving the sign for signed operations
- **Word offset scaling**: Multiplies immediate values by 2 for word-aligned address calculations in load/store operations
- **Immediate prefix support**: Concatenates the 12-bit immediate prefix (from IMM instruction) with the 4-bit immediate field to form 16-bit constants

Program Counter and Branch Logic

The PC management logic handles instruction sequencing:

- **PC register**: Holds the address of the current instruction being executed
- **PC increment**: Normally advances by 2 (word size) to fetch the next sequential instruction
- **Branch unit**: Evaluates condition codes against branch conditions and calculates target addresses for taken branches using sign-extended displacement
- **Jump logic**: Computes jump targets for JAL (jump-and-link) instructions using the ALU result
- **Interrupt vector**: Redirects PC to the interrupt vector address when an interrupt is acknowledged

Memory Interface Logic

The memory interface manages data transfers between the processor and memory:

- **ROM interface**: Fetches instructions from the dual-port ROM (high and low bytes) using the PC as the address
- **BRAM interface**: Handles load and store operations to the data RAM, with separate control for high and low byte banks
- **Byte/word control**: Generates appropriate write enables and output enables for byte (LB/SB) versus word (LW/SW) operations
- **Address parity**: Determines which byte lane to access based on the LSB of the effective address

Peripheral Interface

The peripheral interface provides connectivity to memory-mapped I/O devices:

- **Address decoder**: Detects accesses to peripheral address space (0x1000-0x12FF) and generates appropriate select signals
- **Data routing**: Multiplexes data between the processor data bus and peripheral data buses
- **Ready synchronization**: Coordinates multi-cycle peripheral accesses using ready signals from each peripheral

Interrupt Controller (Enc)

The interrupt encoder prioritizes and manages interrupt requests:

- **Interrupt detection**: Monitors interrupt request signals from peripherals (Timer, UART, Pario)
- **Priority encoding**: Determines which interrupt to service when multiple requests are pending
- **Vector generation**: Provides the appropriate interrupt vector table address to redirect the PC
- **Interrupt acknowledgment**: Coordinates with the Control Unit to insert the interrupt call instruction at appropriate instruction boundaries

Peripheral Modules

Three peripheral modules are integrated into the datapath:

- **UART**: Provides serial communication capability with configurable baud rate, supporting both transmit and receive operations with interrupt generation on receive data available or transmit buffer empty
- **Timer**: A programmable 16-bit counter/timer that can operate in timer mode (counting clock cycles) or counter mode (counting external events), generating periodic interrupts for real-time task scheduling
- **Pario (Parallel I/O)**: An 8-bit parallel input/output port for interfacing with external digital devices, with separate input and output registers accessible through memory-mapped control registers
- **I^2C**: Implements the Inter-Integrated Circuit protocol in master mode only, supporting start/stop condition generation, slave address transmission, and byte-level data transfer over a two-wire serial interface

3.6.3 Data Flow Through the Datapath

During typical instruction execution, data flows through the datapath as follows:

1. The PC addresses ROM to fetch the next instruction
2. The Control Unit decodes the instruction and generates control signals
3. The register file outputs operand values based on source register specifiers
4. Immediate values are formed and sign-extended as needed

5. Operand multiplexers select between register and immediate values
6. The ALU performs the specified operation on the operands
7. The result multiplexer selects the appropriate result (ALU output, memory data, or PC value)
8. The result is written back to the destination register or memory
9. The PC is updated for the next instruction (sequential, branch, or jump)

This pipelined-style data flow, combined with the single-cycle block RAM access, enables efficient instruction execution with minimal stall cycles, achieving good performance despite the non-pipelined architecture.

3.7 Control Unit

The Control Unit serves as the central orchestrator of the processor, responsible for decoding instructions and generating the appropriate control signals that govern the operation of the datapath. As illustrated in Figure 3.10, the Control Unit acts as the interface between the instruction stream and the execution resources, translating opcodes and instruction fields into the specific control signals required for each operation.

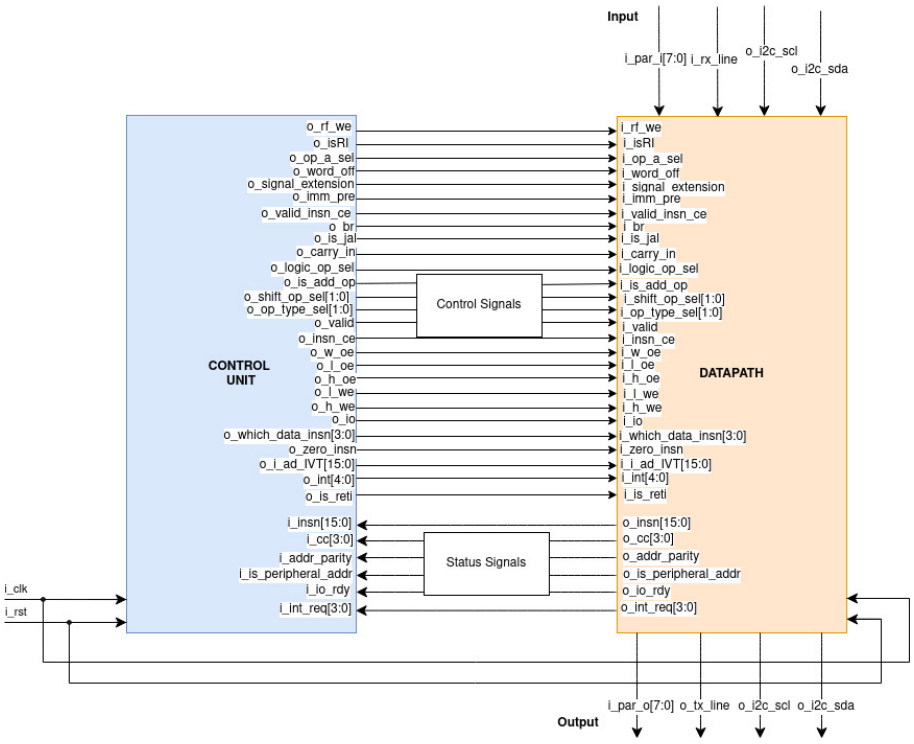

Figure 3.10: Control Unit architecture showing input instruction decoding and output control signals

The Control Unit receives instructions from the on-chip ROM through the instruction fetch interface. The primary input for instruction decoding is:

- **i_insn[15:0]**: The 16-bit instruction word fetched from ROM, containing the opcode, register specifiers, immediate values, and function codes that determine the operation to be executed

Additionally, the Control Unit interfaces with external systems through:

- **i_par_i[7:0]**: Parallel input used for interrupt handling and external communication
- **i_rx_line**: UART receive line for serial communication
- $\mathbf{i}_i 2c\_scl : I2C serial clock line for I2C communication$
- $\mathbf{i}_i 2c\_sda : I2C serial data line for I2C communication$

Based on the instruction word and system status, the Control Unit performs instruction decoding and generates two categories of outputs: control signals that configure the datapath for execution, and status signals that provide feedback to the system.

3.7.1 Control Signals

The control signals generated by the Control Unit configure the datapath for the execution of each instruction. These signals are distributed throughout the datapath to enable specific operations and control data flow. The major control signal groups include:

Register File Control

- **o_rf_we / i_rf_we**: Register file write enable, determines when results are written back to the register file
- **o_isRI / i_isRI**: Indicates an RI-format instruction for proper operand selection

Operand Selection and Data Path Control

- **o_op_a_sel / i_op_a_sel**: Selects source for operand A (register vs. immediate)
- **o_word_off / i_word_off**: Indicates word offset addressing mode
- **o_signal_extension / i_signal_extension**: Controls sign extension of immediate values
- **o_imm_pre / i_imm_pre**: Immediate prefix flag for extended immediate values
- **o_which_data_insn[3:0] / i_which_data_insn[3:0]**: Selects which data source drives the result bus- load byte/word or store byte/word
- **o_zero_insn / i_zero_insn**: Forces instruction to zero (used in interrupt handling)

ALU Operation Control

- **o_valid_insn_ce / i_valid_insn_ce**: Valid instruction clock enable
- **o_br / i_br**: Branch instruction indicator
- **o_is_jal / i_is_jal**: Jump-and-link instruction flag
- **o_carry_in / i_carry_in**: Carry input for add-with-carry operations
- **o_logic_op_sel / i_logic_op_sel**: Selects logical operation (AND, XOR)
- **o_is_add / i_is_add**: Indicates addition operation
- **o_shift_op_sel[1:0] / i_shift_op_sel[1:0]**: Selects shift operation type
- **o_op_type_sel[1:0] / i_op_type_sel[1:0]**: General operation type selector

Memory Access Control

- **o_w_oe / i_w_oe**: Word output enable for load operations
- **o_l_oe / i_l_oe**: Low byte output enable
- **o_h_oe / i_h_oe**: High byte output enable
- **o_l_we / i_l_we**: Low byte write enable for store operations
- **o_h_we / i_h_we**: High byte write enable for store operations
- **o_io / i_io**: I/O operation indicator for peripheral access

Special Instructions

- **o_is_reti / i_is_reti**: Return from interrupt instruction flag
- **o_I_ad_IVT[15:0] / i_I_ad_IVT[15:0]**: Interrupt vector table address
- **o_int[4:0] / i_int[4:0]**: Interrupt request signals

3.7.2 Status Signals

In addition to control signals, the Control Unit also monitors status feedback from the datapath to make informed decoding decisions for subsequent instructions:

- **i_insn[15:0] / o_insn[15:0]**: Current instruction word being decoded
- **i_cc[3:0] / o_cc[3:0]**: Condition codes (Zero, Negative, Carry, Overflow) from ALU operations
- **i_addr_parity / o_addr_parity**: Address parity for byte-addressing alignment
- **i_is_peripheral_addr / o_is_peripheral_addr**: Indicates if current address targets peripheral space
- **i_io_rdy / o_io_rdy**: I/O ready signal from peripheral devices
- **i_int_req[3:0] / o_int_req[3:0]**: Interrupt request status from peripherals

3.7.3 External Interface

The Control Unit also interfaces with the external system through dedicated clock and reset signals:

- **i_clk**: System clock input that synchronizes all control signal generation
- **i_rst**: System reset that initializes the Control Unit to a known state

Additionally, output ports are provided for external visibility:

- **o_par_o[7:0]**: Parallel output for external communication
- **o_tx_line**: Transmit line for serial output
- **o_i2c_scl**: I2C serial clock output line
- **o_i2c_sda**: I2C serial data bidirectional line

3.7.4 Control Unit Operation

The Control Unit operates in a synchronous manner, receiving new instructions on each clock cycle when enabled. The decoding process involves:

1. **Instruction Reception**: Capturing the instruction word from the instruction inputs
2. **Opcode Extraction**: Parsing the instruction format to identify the operation type
3. **Field Decoding**: Extracting register specifiers, immediate values, and function codes
4. **Control Signal Generation**: Activating the appropriate combination of control signals based on the decoded instruction
5. **Status Monitoring**: Evaluating condition codes and ready signals to handle conditional operations and multi-cycle instructions

The modular design of the Control Unit, with clearly defined control and status signal groups, facilitates easy modification and extension of the instruction set. New instructions can be added by implementing additional decoding logic and generating the corresponding control signal combinations, without requiring extensive redesign of the existing control structure.

This separation of control from execution (datapath) exemplifies the classical RISC design philosophy, where a simple, regular control unit generates straightforward control signals for a streamlined datapath, enabling high clock frequencies and efficient FPGA implementation.

3.8 Peripherals

3.8.1 Timer

The timer peripheral was refactored to improve configurability, clarity, and alignment with memory-mapped I/O conventions. In the refactored design, all timer functionality is controlled through a set of dedicated configuration registers, allowing software to manage the timer behavior in a structured and predictable manner.

The main configuration register, **CR#0**, is mapped to address `0x1000`. This register is responsible for controlling the operational mode of the timer. Through specific bit fields, it allows enabling or disabling interrupts, selecting between **timer mode** and **counter mode**, and starting or stopping the timer. The bit assignments follow the specification implemented in the hardware: bit 0 enables interrupt generation on overflow, bit 1 selects between timer and counter operation, and bit 2 controls the start or stop state of the timer. This centralized configuration simplifies software control and replaces the more fragmented control logic present in the original design. Figure 3.11 illustrates the layout of the refactored control register.

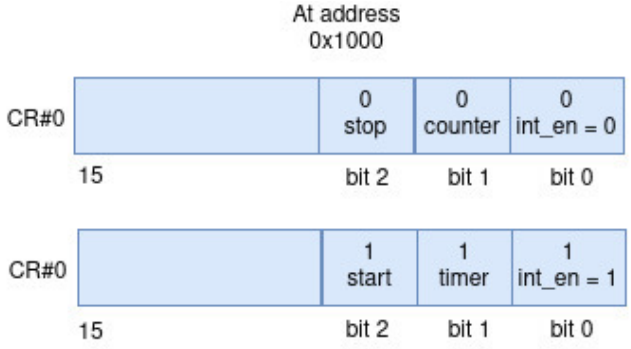

Figure 3.11: Refactored timer control register CR#0 at address 0x1000

The interrupt request register, **CR#1**, is located at address `0x1002`. This register maintains compatibility with the original implementation by asserting the interrupt request bit when a counter or timer overflow occurs, provided that interrupts are enabled. The interrupt request remains asserted until it is explicitly cleared by a write operation to this register. This behavior ensures deterministic interrupt handling and prevents spurious retriggering. Figure 3.12 shows the refactored interrupt request register.

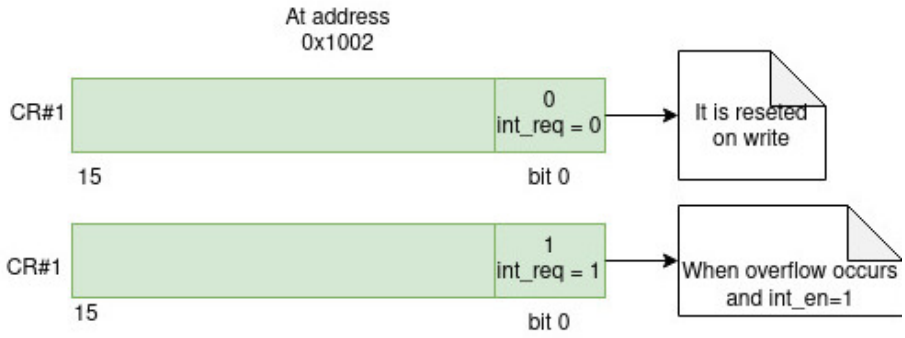

Figure 3.12: Interrupt request register CR#1 at address 0x1002

An additional register, **CR#2**, was introduced at address `0x1004` to allow dynamic configuration of the initial counter value. This 16-bit register defines the value loaded into the counter when it is initialized or reset. If the timer is not running, any write to this register immediately updates the counter value on the next clock cycle. However, if the timer is already running, the new initialization value is only applied after the next overflow event, ensuring consistent operation without disrupting an active counting process. The structure of this register is shown in Figure 3.13.

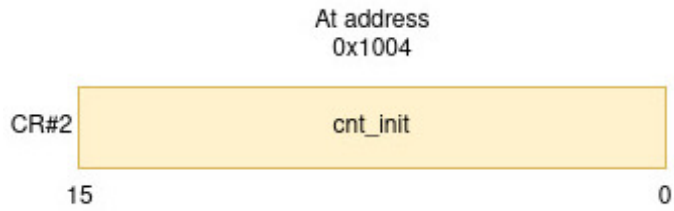

Figure 3.13: Counter initialization register CR#2 at address 0x1004

3.8.2 UART Peripheral

This section describes the design and operation of a memory-mapped Universal Asynchronous Receiver/Transmitter (UART) peripheral. The module provides full-duplex asynchronous serial communication with programmable baud rate, transmit and receive buffering, and interrupt generation.

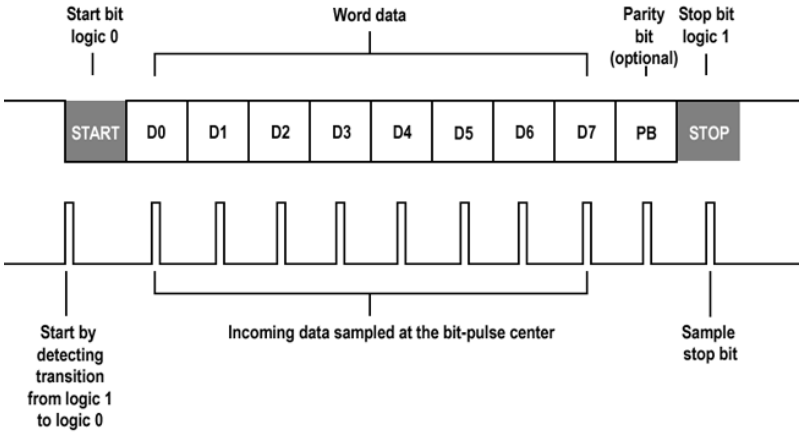

Figure 3.14: UART Frame

The UART peripheral is internally divided into two independent finite-state machines:

- UART Transmitter (TX FSM)
- UART Receiver (RX FSM)

Both FSMs operate concurrently and are synchronized to the system clock.

Peripheral Overview

The UART peripheral supports:

- Configurable baud rate through a programmable divider
- 8-bit data frames
- One start bit and one stop bit
- Full-duplex operation
- Transmit and receive interrupt signaling

Register Map

The UART exposes a set of memory-mapped Special Function Registers (SFRs) summarized in Table 3.2.

Register	Address	Access	Description
SBUF_RX	0x1200	Read Only	Receive buffer register
SBUF_TX	0x1201	Write Only	Transmit buffer register
BAUDRATE_DIV	0x1203	Write Only	Baud rate divider
SCON	0x1205	Write Only	Serial control flags (TI, RI)

Table 3.2: UART Register Map

Register Specification

SBUF_RX
The SBUF_RX register holds the last byte received by the UART receiver. The register is updated by hardware when a valid UART frame is received.

Bit	Name	Access	Reset	Description
15:8	–	–	0	Reserved. Read as zero.
7:0	RX_DATA	RO	0	Received data byte. Contains the last valid byte received by the UART receiver. Valid when the RI flag in SCON is set.

Table 3.3: UART SBUF_RX Register Specification

SBUF_TX

The SBUF_TX register is used to load data for transmission. Writing to this register initiates a UART transmit operation.

Bit	Name	Access	Reset	Description
15:8	–	–	0	Reserved. Ignored by hardware.
7:0	TX_DATA	WO	0	Transmit data byte. Writing a value to this field loads the transmit shift register and starts transmission.

Table 3.4: UART SBUF_TX Register Specification

BAUDRATE_DIV

The BAUDRATE_DIV register defines the UART bit period relative to the system clock. The value is shared by both the transmitter and receiver.

Bit	Name	Access	Reset	Description
15:0	BAUD_DIV	WO	Configurable	Clock divider value defining the UART baud rate. Each UART bit spans one full divider period.

Table 3.5: UART BAUDRATE_DIV Register Specification

By default, the Baudrate value is set to 115200bps.

SCON

The Serial Control (SCON) register is used to report the status of the UART transmitter and receiver. It contains interrupt flags that indicate the completion of transmit and receive operations. The register is memory-mapped and written by hardware upon event completion.

Bit	Name	Access	Reset	Description
7:2	–	–	0	Reserved. Read as zero.
1	TI	HW set	0	Transmit Interrupt flag. Set by hardware when a transmission completes (after the stop bit). This flag can be used to trigger an interrupt or indicate that the transmitter is ready for a new byte.
0	RI	HW set	0	Receive Interrupt flag. Set by hardware when a full byte has been successfully received and stored in the receive buffer. This flag indicates that valid data is available for software to read.

Table 3.6: UART SCON Register Specification

The RI and TI flags are asserted by hardware and may be cleared by software after servicing the corresponding interrupt. Reserved bits must be written as zero and are ignored by the hardware.

Baud Rate Generation

The baud rate is derived from the system clock using a programmable divider. Each UART bit occupies one full baud period. The same divider value is shared between the transmitter and receiver to ensure synchronized timing.

UART Transmitter (TX FSM)

The UART transmitter converts parallel data into a serial bitstream. Transmission is initiated by a write operation to the transmit buffer register.

TX Frame Format Each transmitted frame consists of:

- One start bit (logic 0)
- Eight data bits (least significant bit first)
- One stop bit (logic 1)

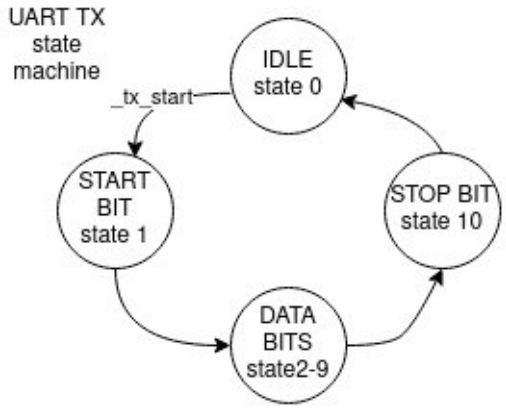

Figure 3.15: UART TX Finite State machine

TX State Machine Description The TX FSM operates according to the flowchart shown in Figure 3.15. The FSM progresses through the following phases:

1. Idle state with TX line held high
2. Start bit transmission
3. Sequential transmission of data bits
4. Stop bit transmission
5. Return to idle and interrupt generation

TX Operation (Pseudocode)

Pseudocode: UART Transmit

```
wait until data is written to TX buffer
load data into shift register
drive TX low for start bit
for each data bit:
    drive TX with current bit
    wait one baud period
drive TX high for stop bit
signal transmit complete (TI)
return to idle
```

UART Receiver (RX FSM)

The UART receiver samples the incoming serial data and reconstructs the transmitted byte. It uses **mid-bit sampling** to improve robustness against timing skew.

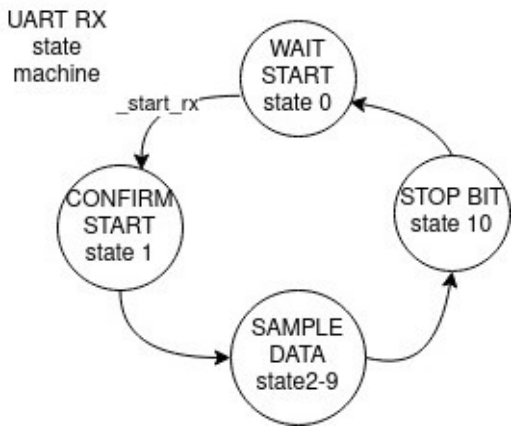

Figure 3.16: UART RX Finite State machine

RX State Machine Description The RX FSM follows the flowchart shown in Figure 3.16. The FSM operates as follows:

1. Idle state monitoring for a start bit
2. Start bit validation using **half-baud delay**
3. Sequential sampling of data bits
4. Stop bit verification
5. Data ready signaling and interrupt generation

RX Operation (Pseudocode)

Pseudocode: UART Receive

```
monitor RX line for falling edge
wait half baud period
if start bit is valid:
    for each data bit:
        wait one baud period
        sample RX and store bit
    wait one baud period
    validate stop bit
    store received byte
    signal receive complete (RI)
return to idle
```

Interrupt Handling

The UART peripheral provides two interrupt sources:

- **RI** (Receive Interrupt): Asserted when a valid byte is received
- **TI** (Transmit Interrupt): Asserted when a transmission completes

These flags allow software to service the UART asynchronously without polling.

Ready Signal

The `o_rdy` signal indicates that the UART peripheral is selected and able to respond to control bus transactions.

Summary

The UART peripheral implements a compact and modular asynchronous serial interface using independent transmit and receive state machines. The use of flowchart-defined FSMs simplifies verification and provides a clear correspondence between the specification and implementation.

3.8.3 I²C Master Peripheral

This section describes the design and operation of a memory-mapped I²C master peripheral. The module implements a multi-speed I²C controller supporting Standard, Fast, Fast-Plus, High-Speed, and Ultra-Fast modes through a programmable clock divider. The peripheral is controlled via a generic control bus and operates as the bus master.

Peripheral Overview

The I²C master provides:

- Programmable serial clock (SCL) frequency
- Start and stop condition generation
- 7-bit addressing with read/write support
- Byte-level transmit and receive
- ACK/NACK handling

The serial data (SDA) and serial clock (SCL) lines are implemented as open-drain bidirectional signals, compliant with the I²C specification.

Register Map

The peripheral exposes a small set of memory-mapped registers summarized in Table 3.7.

Register	Address	Access	Description
I2C_DIVIDER	0x1300	Write Only	Clock divider for SCL generation
I2C_DATA	0x1302	Write Only	Address and data payload register
I2C_CONTROL	0x1304	Write Only	Control bits (Enable, Start)

Table 3.7: I²C Register Map

Register Specification

I2C_DIVIDER Register Specification

The I2C_DIVIDER register controls the serial clock (SCL) frequency of the I²C master by dividing the system clock.

Bit	Name	Access	Reset	Description
15:0	I2C_DIV	WO	0x01F4	Clock divider value used to generate the I²C SCL signal. Determines the communication speed.

Table 3.8: I²C I2C_DIVIDER Register Specification

After reset, this register configures II²C communication at 100kbps.

I2C_DATA Register Specification

Bit	Name	Access	Reset	Description
15:8	DATA	WO	0	Data byte to be transmitted or buffer for received data during read operations.
7:0	ADDR	WO	0	7-bit slave address with the least significant bit indicating read/write operation.

Table 3.9: I^{2}C I2C_DATA Register Specification

The I2C_DATA register stores the address and data payload used during an I^{2}C transaction.

I2C_CONTROL Register Specification

The I2C_CONTROL register controls the operation of the I^{2}C master state machine.

Bit	Name	Access	Reset	Description
7:2	–	–	0	Reserved. Must be written as zero.
1	START	WO	0	Start command. When set, initiates an I^{2}C transaction. Cleared internally after execution.
0	EN	WO	0	Enable bit. Enables the I^{2}C master state machine when set.

Table 3.10: I^{2}C I2C_CONTROL Register Specification

Clock Generation

The I^{2}C clock is derived from the system clock using a programmable divider. The divider value determines the SCL frequency and allows selection of multiple I^{2}C speed modes at compile time or runtime.

Internally, the divider generates:

- A toggling internal I^{2}C clock
- Rising-edge enable signals for data sampling
- Falling-edge enable signals for data driving

SCL is only actively driven during address and data phases; it remains high during idle, start, and stop conditions.

I^{2}C State Machine

The protocol is implemented using a finite-state machine synchronized to the internal I^{2}C clock. The main states are:

1. **IDLE**: Bus is idle; SCL and SDA are high
2. **START**: Start condition generation
3. **ADDRESS**: Transmission of address and R/W bit
4. **READ_ACK**: ACK sampling from slave
5. **WRITE_DATA**: Byte transmission to slave
6. **WRITE_ACK**: Master ACK after read

7. **READ_DATA**: Byte reception from slave
8. **STOP**: Stop condition generation

Protocol Operation (Pseudocode)

The high-level behavior of the I^{2}C master can be summarized as follows:

Pseudocode: I^{2}C Transaction

```
wait until ENABLE is set
if START requested:
    generate START condition
    transmit address + R/W bit
    if slave ACK received:
        if WRITE operation:
            transmit data byte
            check ACK
        else if READ operation:
            receive data byte
            transmit ACK
    generate STOP condition
return to IDLE
```

Data Direction and SDA Control

The SDA line is controlled dynamically based on the current protocol phase:

- During address and write phases, SDA is actively driven by the master
- During ACK and read phases, SDA is released and sampled

This behavior ensures proper open-drain operation and allows slave devices to acknowledge or supply data as required.

Transaction Completion

A transaction completes when the STOP condition is generated and the state machine returns to the IDLE state. At this point, the peripheral is ready to accept a new START command.

Summary

The I^{2}C master peripheral provides a compact and configurable implementation of the I^{2}C protocol. Its register-based control interface enables easy software integration, while the internal state machine ensures protocol-compliant timing and sequencing across multiple speed modes.

3.9 Interrupt Controller Architecture

The Refactored GR0040 implements an interrupt structure inspired by the 8051 microcontroller architecture. This design provides a clear, simple, and standardized organization for handling interrupt triggers without the need for complex software-level lookups or indirections.

The interrupt process is managed through a hardwired sequence consisting of four primary stages:

1. **Interruption Prologue:** A hardwired sequence initiated upon an interrupt trigger.
2. **Vector Jump:** The Program Counter (PC) automatically jumps to a fixed address corresponding to the specific interrupt source in the Interrupt Vector Table (IVT).
3. **ISR Execution:** The processor executes the Interrupt Service Routine (ISR) handler.
4. **Interruption Epilogue:** A hardwired sequence that restores the PC to its pre-interruption state, allowing normal program execution to resume.

3.10 Interrupt Vector Table (IVT)

The IVT defines fixed entry points for various interrupt sources. Each maskable interrupt handler is allocated 8 words (16 bytes), allowing for a maximum of 8 instructions within the vector itself. If a handler requires more space, it must include a jump instruction to a different memory location where the remainder of the code resides.

Table 3.11: GR0040 Interrupt Vector Table

Interrupt Source	**ISR Address (Hex)**	**Description**
Reset	0x0002	Reset Vector (Jumps to 0x0054)
Timer	0x0004	Timer/Counter Interrupt
Extern 0	0x0014	External Interrupt Line 0
Extern 1	0x0024	External Interrupt Line 1
UART_Rx	0x0034	Signals Reception on UART
UART_Tx	0x0044	Signals Transmission on UART

3.11 Implementation and Priority Logic

The interrupt system is implemented using a priority-based "If-Else" logic structure within the Control Unit. During every positive clock edge, the system verifies the interrupt request lines (Figure 3.17).

- **Priority Handling:** Interrupts are processed based on a hardwired priority. If multiple requests occur simultaneously, the "Priority If-Else" logic determines which source is serviced first.
- **Jump-and-Link:** The jump to the ISR and the linking process (saving the return address) are entirely hardwired.
- **Address Space:** Interrupt handling and program data are mapped within the first 1 kB of on-chip RAM (BRAM), specifically in the range 0x0000 to 0x03FF.

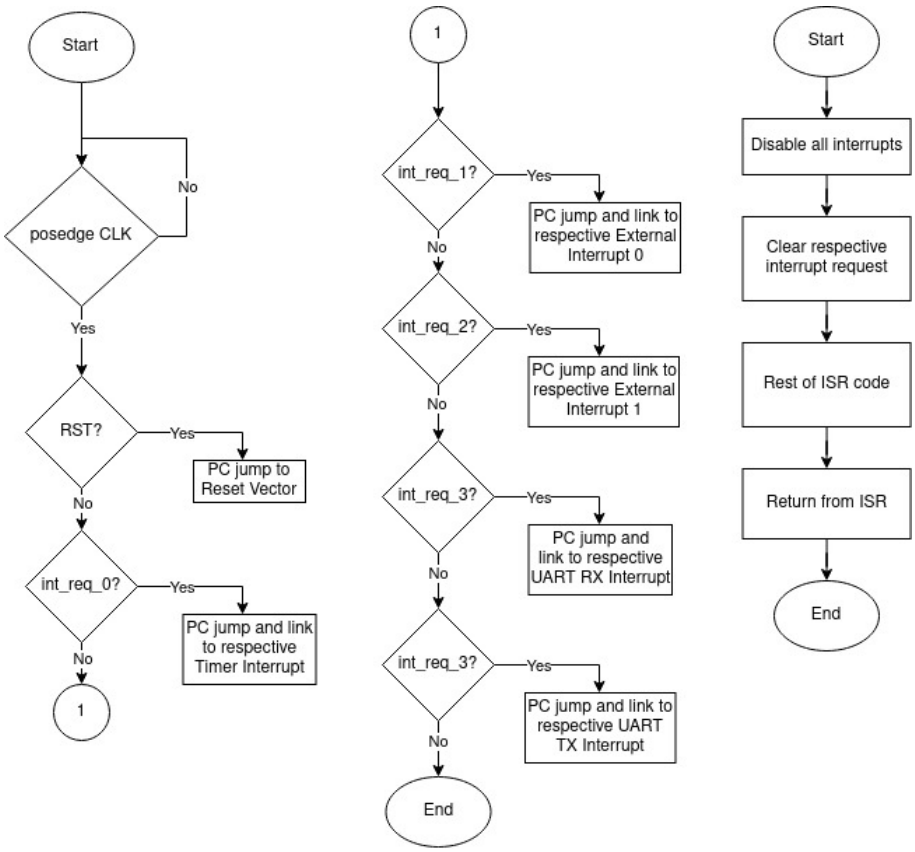

Figure 3.17: Interrupt controller flowchart

3.11.1 Priority Encoding

The interrupt handling mechanism implemented in this processor draws inspiration from the classical 8051 microcontroller architecture, particularly in its priority-based interrupt servicing approach. Similar to the 8051, which supports multiple interrupt sources with fixed priority levels, this design implements a priority encoder that determines which interrupt request should be serviced when multiple interrupts occur simultaneously.

Comparison with 8051 Interrupt Architecture

The 8051 microcontroller features a two-level priority interrupt system with no indirections with five interrupt sources, each with a fixed priority. When multiple interrupts are pending, the 8051's interrupt controller automatically services the highest priority interrupt first. Our implementation extends this concept with the following key similarities and enhancements:

- **Priority-based servicing**: Like the 8051, interrupts are assigned fixed priorities based on their source, with lower-indexed interrupts having higher priority
- **Interrupt queuing**: Pending interrupts are queued while an interrupt service routine is executing, similar to the 8051's interrupt pending flags
- **Non-maskable execution**: Once an interrupt begins servicing, it runs to completion (until RETI) before another interrupt can be serviced, preventing interrupt nesting
- **Atomic instruction protection**: Certain instruction sequences cannot be interrupted, maintaining data integrity during critical operations

Interrupt Priority Encoding Design

The priority encoding logic manages interrupt requests through several coordinated stages, as illustrated in Figure 3.19. The implementation ensures that interrupts are serviced fairly while respecting priority levels and protecting critical instruction sequences.

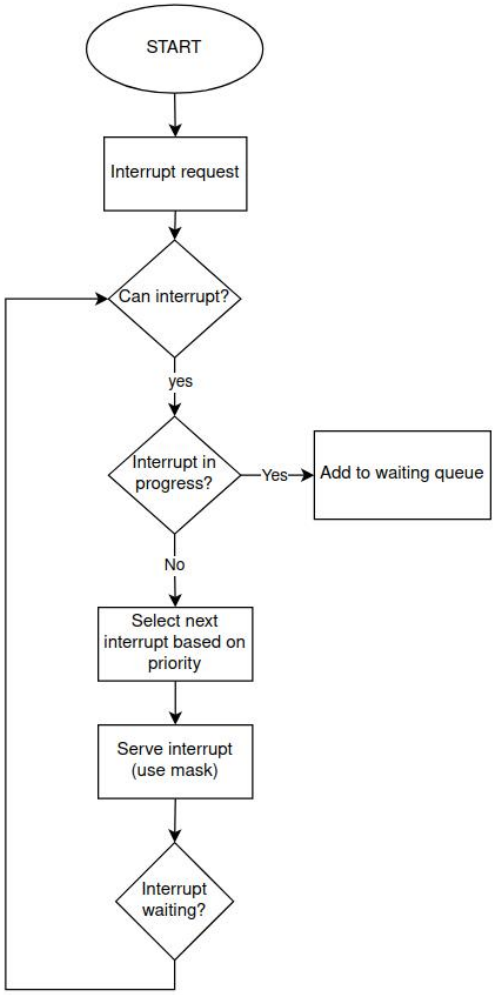

Figure 3.18: Priority Encoding flowchart

Timer/Counter Interrupt Example

The timer peripheral (assigned to address space 0x1000–0x10FF) utilizes Interrupt Line 0. When a timer overflow occurs:

1. PC is saved by hardaware.
2. The system jumps to address 0x0004 (Timer ISR).
3. The `int_req` flag is cleared by hardware (on UART interrupts, the bits must be cleared by software).
4. The handler code is executed.
5. The handler executes a return from interrupt routine (RETI) to return to the original program flow.

Overview

Figure 3.19 showcases the different implemented instructions for each ISR. Firstly, regarding the timer interrupt, on the timer overflow, this interrupt is triggered. In its ISR a virtual timer is created so a board LED can blink.

Secondly, the external interrupts are used to start and stop the timer and, therefore, we can make the LED stop blinking and resume blinking. Code execution enters in each of these interrupts when a falling edge is detected on its respective pin.

Additionally, the UART_Rx interrupt aims to implements an "echo" algorithm, where date receive (Rx) is echoed to the transmission line (Tx). Program execution enters here after UART recepetion has been detected.

Lastly, the program enters in the UART_Tx interrupt after sending a character via UART. The implemented interrupt service routine toggles a board LED and clears the transmission flag (TI).

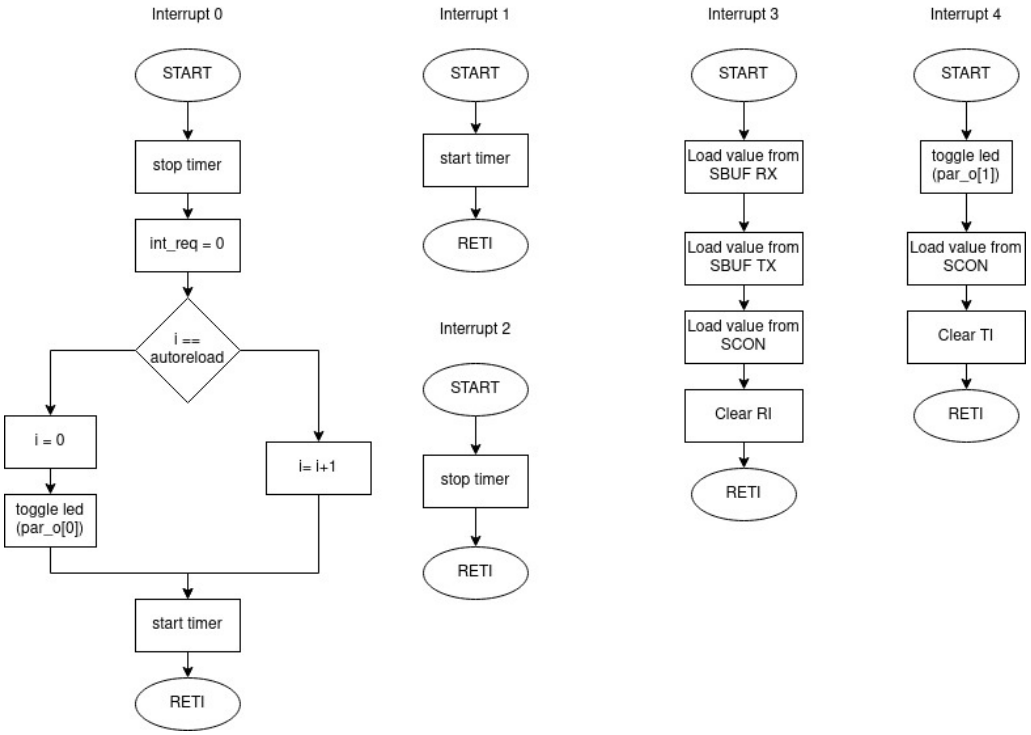

Figure 3.19: ISRs flowchart for each interrupt

3.12 Latch Avoidance

The inference of latches must be avoided and is explicitly warned against by Vivado. Inferred latches are typically the result of improperly specified combinational logic, where not all possible input conditions are handled. When a signal is not assigned a value in every execution path of a combinational process, the synthesis tool preserves the previous value, which leads to latch inference.

Inferred latches are undesirable because they may introduce unintended state retention, unpredictable behavior, and potential glitches. Furthermore, the presence of latches complicates timing analysis and significantly increases the difficulty of debugging and verification, particularly in large or deeply pipelined designs.

3.12.1 Understanding Latch Formation

To understand why latches form and how glitches arise, consider the example shown in Figure 3.20. The circuit implements a simple logic function where:

- Input (A) = 0
- Input (B) transitions from 1 to 0
- Input (C) = 1

As illustrated in the timing diagram, when signal B transitions, the intermediate signals AB (output of the top NAND gate) and BC (output of the bottom NAND gate) experience different propagation delays. This creates a brief moment where both AB and BC are temporarily high, causing the output Y to glitch from 1 to 0 and back to 1. This glitch occurs because the combinational logic path has not been fully specified for all input transitions, allowing transient states to propagate through the circuit.

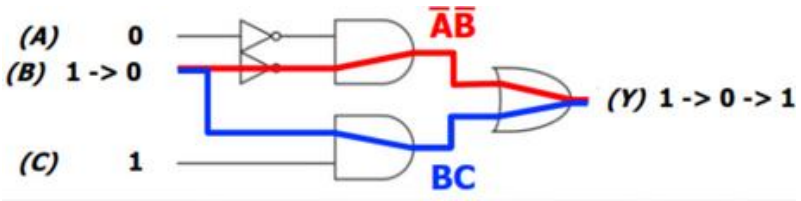

Figure 3.20: Glitch formation in combinational logic due to propagation delay differences. The output Y experiences an unwanted transition (1→0→1) when input B changes, demonstrating the hazard that can occur in incompletely specified logic.

When such incompletely specified combinational logic is synthesized, the tools may infer a latch to "hold" the previous value during undefined states. Figure 3.21 illustrates this concept using a Karnaugh map representation. The function $Q^+(C, D, Q)$ represents the next state logic, where:

- $e1 = \overline{C} \cdot Q$ represents one term (circled in blue)
- $e2 = C \cdot D$ represents another term (highlighted in pink)
- The complete expression is $Q^+(C, D, Q) = e1 + e2 = \overline{C} \cdot Q + C \cdot D$

The critical observation is that the current value of Q appears in the next-state equation. This feedback of the current state into the next state is the defining characteristic of a latch. In the Karnaugh map, the cells where Q appears in the expression (shown with value 1 in the blue circled region when C=0) demonstrate that the circuit must "remember" its previous value, thus requiring storage—a latch.

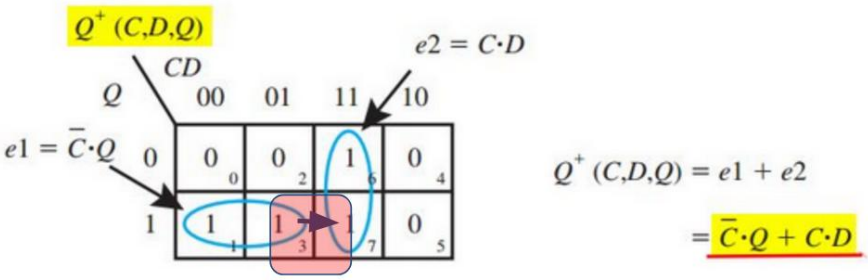

Figure 3.21: Karnaugh map showing latch inference when output depends on its previous value. The expression $Q^+(C, D, Q) = \overline{C} \cdot Q + C \cdot D$ contains the current state Q, requiring a storage element (latch) for implementation. The overlapping regions (e1 and e2) show how incomplete specification leads to state retention.

A common but incorrect workaround is the addition of feedback or self-assignments intended to "hold" the previous value of a signal. While such constructs may silence

synthesis warnings in some cases, they do not eliminate the underlying issue. Instead, they explicitly describe storage behavior by relying on the previous value of the signal, which by definition results in latch inference due to the absence of complete assignment specifications.

For example, code like:

```
always @(*) begin
    if (condition)
        signal = new_value;
    else
        signal = signal;  // WRONG: Creates latch
end
```

This explicitly creates a feedback loop where the signal depends on its own previous value, exactly as shown in the Karnaugh map of Figure 3.21.

Proper Latch Avoidance Strategy

To properly avoid inferred latches, all combinational processes were refactored to ensure that every output signal is assigned a well-defined value for all possible input conditions. This was achieved by:

1. **Providing default assignments**: At the beginning of each combinational process, all outputs are assigned default values
2. **Exhaustive case coverage**: All branches of conditional statements (if-else, case) are explicitly handled
3. **Complete sensitivity lists**: Using `always @(*)` or equivalent to ensure all read signals are in the sensitivity list
4. **Eliminating feedback**: Ensuring no output signal depends on its own previous value in combinational logic

The correct approach:

```
always @(*) begin
    signal = default_value;  // Default assignment
    if (condition)
        signal = new_value;
    // No else needed - default covers all other cases
end
```

As a result of these refactoring efforts, the logic is fully combinational, deterministic, and free of unintended storage elements. The design now passes synthesis without latch warnings, and the combinational paths are well-defined for all input combinations, eliminating the potential for glitches and unpredictable behavior as demonstrated in Figures 3.20 and 3.21.

3.13 Removal of Internal Tri-State

This refactoring addresses the use of internal tri-state (high-impedance, `Z`) signals within the design. Recent versions of Verilog and modern synthesis tools do not support the use of high-impedance states for internal variables. According to UG901 (Vivado Design Suite User Guide: Synthesis), tri-state signals are intended to be used exclusively for top-level I/O ports. This behavior has also been confirmed through AMD Community Support.

The presence of internal tri-state assignments may lead to non-synthesizable logic, unexpected synthesis results, or implicit logic inference that diverges from the intended hardware behavior. As a consequence, internal signals driven to a high-impedance state

can introduce ambiguity in simulation and are not reliably mapped to physical resources during synthesis.

To resolve this issue, all internal signal assignments were refactored to ensure that signals are always driven to known and deterministic values. In cases where the original intent was to logically disconnect a signal from the circuit (previously modeled using a high-impedance state), an additional control variable was introduced or an existing one was reused. This control logic explicitly manages signal activation and deactivation, preserving the original functional intent without relying on unsupported tri-state behavior.

Chapter 4

Implementation

4.1 ISA refactored

4.1.1 Shift operations

The hardware implementation of the shift operations is achieved through a concise multiplexer-based design, as shown in Listing 4.1. The selection signal `i_shift_op_sel[0]` determines the shift direction (right or left), while `i_shift_op_sel[1]` differentiates between logical and arithmetic right shifts. For right shifts, a logical shift inserts a zero in the most significant bit position, whereas an arithmetic shift preserves the sign bit. Left shifts are identical for both logical and arithmetic variants, simply inserting a zero in the least significant bit position.

```
1 assign o_shift_result =
2     i_shift_op_sel[0] ?                                   // RIGHT shift
3         (i_shift_op_sel[1] ?
4             {1'b0, i_a[`DATA_MSB:1]} :                    // LOGICAL
                  right
5             {i_a[`DATA_MSB], i_a[`DATA_MSB:1]}) :         // ARITHMETIC
                  right
6         {i_a[`DATA_MSB-1:0], 1'b0};                       // LEFT shift
```

Listing 4.1: Shift operation implementation

In the following example, it is possible to observe that a **shift left logical (SLL)** operation was performed. The original value 0003 becomes 0006 after the execution of this instruction, demonstrating that the value was effectively multiplied by 2.

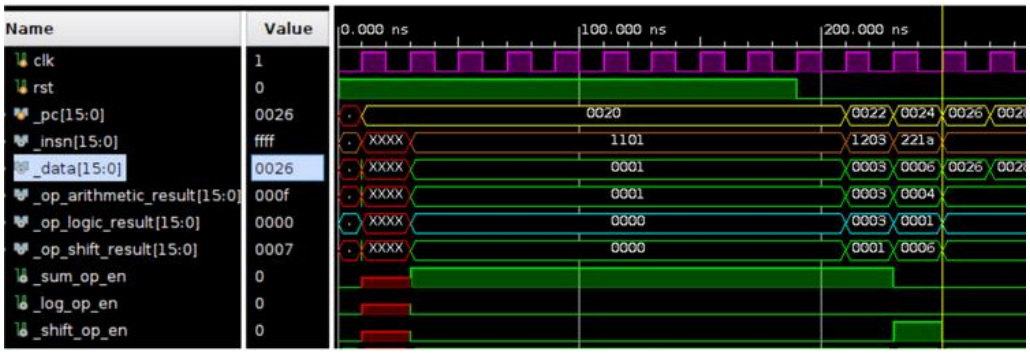

Figure 4.1: Example of a Shift Left Logical (SLL) instruction: the value 0003 is shifted left, resulting in 0006.

4.1.2 RETI instruction

The implementation, shown in Listing 4.2, consists of two main components. In the control unit, the RETI instruction is decoded when the opcode equals 10, and a control signal `o_is_reti` is generated to indicate this instruction type. In the datapath, a dedicated register `_return_addr` stores the instruction register value at the moment an interrupt occurs. The program counter multiplexer logic then selects the appropriate next address: the interrupt vector table address when an interrupt is detected, the stored return address when RETI is executed, or the normal sequential or jump target address otherwise.

```
// In control unit:
`define RETI    (_op==10)
assign o_is_reti = o_valid & `RETI;

// In datapath:
reg [`ADDR_MSB:0] _return_addr;
always @(posedge i_clk) begin
    if (i_rst)
        _return_addr <= 16'h0000;
    else if (|i_int)
        _return_addr <= _IR;
end

assign _PC  = (|i_int) ? i_i_ad_IVT :
              i_is_reti ? _return_addr :
              (i_is_jal) ? _op_arithmetic_result : _IRincd;
```

Listing 4.2: RETI instruction implementation

Figure 4.2 illustrates the correct operation of the `RETI` mechanism. Upon entering an interrupt, the interrupt request line (`i_int`, shown in violet) is asserted, triggering the storage of the current instruction register value into `_return_addr`. Later, when the `RETI` instruction is executed to exit the interrupt service routine, the corresponding instruction signal (`o_insn`, shown in orange) is asserted, and the program counter is updated with the previously saved return address. As a result, execution resumes at the instruction immediately preceding the interrupt, validating the correct functionality of the implementation.

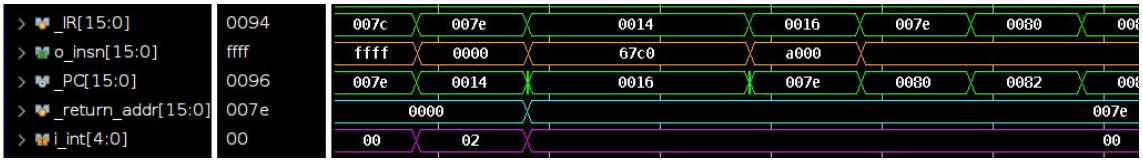

Figure 4.2: RETI instruction working

4.2 Register file

```
regfile16x16 regfile(
    .a(_rd),                        //write addr
    .d(_data),                      //write data
    .dpra(i_isRI ? _rd : _rs),      //dual port read addr
    .clk(i_clk),
    .we(i_rf_we),                   //register file write enable
    .spo(_destinationRegister),     //single port output
    .dpo(_2ndOp)                    //dual port output
  );
```

4.3 Memory

Listing 4.3 shows the Verilog implementation of the separated ROM and RAM architecture. The code clearly demonstrates the division between instruction ROM (high and low bytes) and data RAM (high and low bytes).

```
1
2  ROM_8_512_H romh_insn(
3      .rsta(i_zero_insn),
4      .ena(i_insn_ce),
5      .clka(i_clk),
6      .addra(_PC[9:1]),
7      .douta(o_insn[15:8]));
8
9  ROM_8_512_L roml_insn(
10     .rsta(i_zero_insn),
11     .ena(i_insn_ce),
12     .clka(i_clk),
13     .addra(_PC[9:1]),
14     .douta(o_insn[7:0]));
15
16 BRAM_8_512_H ramh_data(
17     .rsta(i_rst),
18     .wea(i_h_we),
19     .ena(1'b1),
20     .clka(i_clk),
21     .addra(_data_address[9:1]),
22     .dina(_do_h),
23     .douta(_di[15:8]));
24
25 BRAM_8_512_L raml_data(
26     .rsta(i_rst),
27     .wea(i_l_we),
28     .ena(1'b1),
29     .clka(i_clk),
30     .addra(_data_address[9:1]),
31     .dina(_data_to_store[7:0]),
32     .douta(_di[7:0]));
```

Listing 4.3: Separated ROM and RAM implementation with high/low byte organization

This refactored memory architecture provides a more robust, maintainable, and scalable foundation for the system compared to the original unified BRAM approach, while maintaining the performance characteristics critical for embedded applications.

4.4 Datapath and Control Unit

In the following listings, the top-level interconnection between the Control Unit and the Datapath is shown. These two modules form the core of the processor architecture, where the Control Unit is responsible for instruction decoding and control signal generation, while the Datapath executes the operations dictated by those control signals.

The bidirectional exchange of signals between the Control Unit and the Datapath enables a clean separation of concerns and improves modularity. This structure simplifies verification and future extensions of the processor, while ensuring that instruction execution, peripheral access, and interrupt handling are performed in a deterministic and well-defined manner.

```
ControlUnit controlunit (
 .i_clk(_clk50),
 .i_rst(_rst_sys),
 .i_insn(_insn),
 .i_cc(_cc),
 .i_addr_parity(_addr_parity),
 .i_is_peripheral_addr(_is_
 peripheral_addr),
 .i_io_rdy(_io_rdy),
 .i_int_req(_int_req),
 .o_valid_insn_ce(_valid_insn_ce),
 .o_isRI(_isRI),
 .o_op_a_sel(_op_a_sel),
 .o_rf_we(_rf_we),
 .o_word_off(_word_off),
 .o_signal_extension(_signal_
 extension),
 .o_imm_pre(_imm_pre),
 .o_br(_br),
 .o_is_jal(_is_jal),
 .o_carry_in(_carry_in),
 .o_logic_op_sel(_logic_op_sel),
 .o_shift_op_sel(_shift_op_sel),
 .o_op_type_sel(_op_type_sel),
 .o_valid(_valid),
 .o_insn_ce(_insn_ce),
 .o_is_add_op(_is_add_op),
 .o_w_oe(_w_oe),
 .o_l_oe(_l_oe),
 .o_h_oe(_h_oe),
 .o_l_we(_l_we),
 .o_h_we(_h_we),
 .o_io(_io),
 .o_which_data_insn(_which_
 data_insn),
 .o_zero_insn(_zero_insn),
 .o_i_ad_IVT(_i_ad_IVT),
 .o_int(_int),
 .o_is_reti(_is_reti)
);
```

Listing 4.4: Control Unit instantiation

```
Datapath datapath (
 .i_clk(_clk50),
 .i_rst(_rst_sys),
 .i_rf_we(_rf_we),
 .i_isRI(_isRI),
 .i_op_a_sel(_op_a_sel),
 .i_word_off(_word_off),
 .i_signal_extension(_signal_
 extension),
 .i_imm_pre(_imm_pre),
 .i_valid_insn_ce(_valid_insn_ce),
 .i_br(_br),
 .i_is_jal(_is_jal),
 .i_carry_in(_carry_in),
 .i_logic_op_sel(_logic_op_sel),
 .i_is_add_op(_is_add_op),
 .i_shift_op_sel(_shift_op_sel),
 .i_op_type_sel(_op_type_sel),
 .i_valid(_valid),
 .i_insn_ce(_insn_ce),
 .i_w_oe(_w_oe),
 .i_l_oe(_l_oe),
 .i_h_oe(_h_oe),
 .i_l_we(_l_we),
 .i_h_we(_h_we),
 .i_io(_io),
 .i_which_data_insn(_which
 _data_insn),
 .i_par_i(w_par_i),
 .i_zero_insn(_zero_insn),
 .i_i_ad_IVT(_i_ad_IVT),
 .i_int(_int),
 .i_rx_line(w_rx_line),
 .i_is_reti(_is_reti),
 .o_insn(_insn),
 .o_cc(_cc),
 .o_addr_parity(_addr_parity),
 .o_is_peripheral_addr(_is_
 peripheral_addr),
 .o_io_rdy(_io_rdy),
 .o_par_o(w_par_o),
 .o_int_req(_int_req),
 .o_tx_line(_tx_line)
);
```

Listing 4.5: Datapath instantiation

4.4.1 Datapath implementation and module instantiation

```
    `timescale 1ns / 1ps

`include "Constants.vh"

`define IO

/************************
 * DATAPATH MODULE
 ************************/

```

```
11  module Datapath(
12      ...
13      );
14
15  /***************************************
16   * INTERNAL VARIABLES
17   ***************************************/
18    reg [`ADDR_MSB:0] _IR;
19    wire [`ADDR_MSB:0] _PC;
20    wire [`ADDR_MSB:0] _data_address;
21    wire [`DATA_MSB:0] _data_to_store;
22    wire [`DATA_MSB:0] _data;
23
24    wire [`DATA_MSB:0] _io_data;
25
26
27  /* INSTRUCTION ARGUMENTS DECODE - DATAPATH */
28    wire [3:0]  _rd = o_insn[11:8];
29    wire [3:0]  _rs = o_insn[7:4];
30    wire [3:0] _imm = o_insn[3:0];
31    wire [11:0] _i12 = o_insn[11:0];
32    wire [7:0] _disp = o_insn[7:0];
33
34
35  /* Select Operation */
36    wire [`DATA_MSB:0] _op_arithmetic_result;
37    wire [`DATA_MSB:0] _op_logic_result;
38    wire [`DATA_MSB:0] _op_shift_result;
39
40  /* ALU operands  */
41    wire [`DATA_MSB:0] _destinationRegister; // Value just Stored on
        the RF
42    wire [`DATA_MSB:0] _2ndOp;    // b selection in RF
43
44  /***************************************
45   * IMMEDIATE OPERATION FORMATION
46   ***************************************/
47
48   reg [11:0] _i12_pre;
49   always @(posedge i_clk)
50      if (i_valid_insn_ce)
51        _i12_pre <= _i12;
52
53    wire _sxi = i_signal_extension & _imm[3];
54    wire [10:0] _sxi11 = {11{_sxi}};
55    wire _i_4 = _sxi | (i_word_off & _imm[0]);
56    wire _i_0 = ~i_word_off & _imm[0];
57    wire [`DATA_MSB:0] _imm16 = i_imm_pre ? {_i12_pre, _imm} :
        {_sxi11, _i_4, _imm[3:1], _i_0};
58
59   /***************************************
60   * Instruction Register (IR) UPDATE
61   ***************************************/
62
63      reg [`ADDR_MSB:0] _return_addr;
64
65      always @(posedge i_clk) begin
66          if (i_rst)
67              _return_addr <= 16'h0000;
68          else if (|i_int)
69              _return_addr <= _IR;
```

```
70      end
71
72    wire [6:0] _sxd7   = {7{_disp[7]}};  //signal extension disp8
          for jump
73    wire [`DATA_MSB:0] _sxd16 = {_sxd7,_disp,1'b0};    //-> result
          of disp8*2  multiply by 2 means shift right and LSb = 0
74    wire [`DATA_MSB:0] _IRinc = i_br ? _sxd16 : {i_valid,1'b0};
          //if hit instruction is the next sequential
75    wire [`DATA_MSB:0] _IRincd = _IR + _IRinc;
76
77    assign _PC  =  (|i_int) ? i_i_ad_IVT :
78                    i_is_reti ? _return_addr :
79                   (i_is_jal) ? _op_arithmetic_result : _IRincd;
80
81    always @(posedge i_clk)
82      if (i_rst)
83       _IR <= i_i_ad_IVT;
84      else if(i_valid_insn_ce)
85        _IR <= _PC;
86
87  /***************************************
88   * ALU
89   ***************************************/
90
91    /* ALU Operand Selection */
92    wire [`DATA_MSB:0] _a = i_op_a_sel ? _destinationRegister :
          _imm16;
93    wire [`DATA_MSB:0] _b = _2ndOp;
94
95    assign _data_address = _op_arithmetic_result;           // SUM
          RESLUT - efective address
96    assign o_addr_parity = _data_address[0];
97    assign o_is_peripheral_addr = _data_address[`IO_IDENTIFIER];
98    assign _data_to_store = _destinationRegister;      //d_ad
          (destination address) is stored here
99
100 /***************************************
101  * STORE/LOAD
102  ***************************************/
103   wire _sw, _sb, _lw, _lb;
104   assign {_lw, _lb, _sw, _sb} = i_which_data_insn;
105
106   wire [`DATA_MSB:0] _di;
107   wire [7:0] _do_h = _sw ? _data_to_store[15:8] :
          _data_to_store[7:0];
108
109   assign _data =
110     o_io_rdy ? _io_data :
111     i_w_oe ? {_di[15:8], _di[7:0]} :
112     i_l_oe ? {8'h00, _di[7:0]} :
113     i_h_oe ? {8'h00, _di[15:8]} :
114
115     (i_op_type_sel == 2'b00) ? _op_arithmetic_result :
116     (i_op_type_sel == 2'b01) ? _op_logic_result      :
117     (i_op_type_sel == 2'b10) ? _op_shift_result      :
118     /*(i_op_type_sel == 2'b11)  */ _IR;
119     // Default value is _IR, given the default value of
          o_op_type_sel in C.U.
120
121 /***************************************
122  * PERIPHERAL CONTROL
```

```
123 ****************************************/
124     `ifdef IO
125 
126         reg [`IO_REG_MSB:0] _io_ad;
127 
128         always @(posedge i_clk) _io_ad <= _data_address;
129 
130         wire [`CONTROL_MSB:0] _ctrl;
131         wire [`SELECT_MSB:0] _sel;
132 
133         wire [`SELECT_MSB:0] _per_rdy; //peripheral ready
134         assign _per_rdy[`SELECT_MSB:3] = {(`SELECT_MSB-3+1){1'b0}};
135         assign o_io_rdy  = | (_sel & _per_rdy);
136 
137      ctrl_enc enc(
138         .i_clk(i_clk), .i_rst(i_rst), .i_io(i_io),
139         .i_io_ad(_io_ad), .i_lw(_lw), .i_lb(_lb), .i_sw(_sw),
140         .i_sb(_sb), .o_ctrl(_ctrl), .o_sel(_sel));
141 
142 
143       wire [`DATA_MSB:0] _timer_data;
144       wire [`DATA_MSB:0] _pario_data;
145       wire [`DATA_MSB:0] _uart_data;
146 
147     assign _io_data = _sel[0] ? {8'h00, _timer_data[`PARIO_MSB:0]}
        :
148                       _sel[1] ? {8'h00, _pario_data[`PARIO_MSB:0]}
                             :
149                       _sel[2] ? {8'h00, _uart_data[`PARIO_MSB:0]}:
150                       16'h0000;
151 
152       timer timer(
153         .i_ctrl(_ctrl), .i_sel(_sel[0]),
154         .i_i(1'b1), .i_data_to_store(_data_to_store),
155         .i_cnt_init(16'h0000), .o_rdy(_per_rdy[0]),
              //0x1000-0x10FF --> 255 endere os
156         .o_data(_timer_data), .o_int_req(o_int_req[0]));
157 
158      pario par(
159        .i_rst(i_rst), .i_ctrl(_ctrl), .i_sel(_sel[1]),
160        .i_i_8bits(i_par_i),
              .i_data_to_store(_data_to_store[`PARIO_MSB:0]),
161        .o_data(_pario_data), .o_rdy(_per_rdy[1]), //0x1100-0x11FF
           --> 255 endere os
162        .o_o_8bits(o_par_o), .o_int_req(o_int_req[2:1]));
             //o_int_req[2:1]
163 
164      uart uart(
165         .i_ctrl(_ctrl), .i_sel(_sel[2]), .i_rx_line(i_rx_line),
166         .i_data_to_store(_data_to_store), .o_rdy(_per_rdy[2]),
              .o_tx_line(o_tx_line),
167         .o_data(_uart_data), .o_int_req(o_int_req[4:3]));
168 
169     `else
170         assign o_int_req = 4'b00000;
171     `endif
172 
173 
174 /****************************************
175  * INTERNAL MODULES
176  ****************************************/
```

```
177
178       regfile16x16 regfile(
179         .a(_rd),                          //write addr
180         .d(_data),                        //write data
181         .dpra(i_isRI ? _rd : _rs),        //dual port read addr
182         .clk(i_clk),
183         .we(i_rf_we),                     //register file write
               enable
184         .spo(_destinationRegister),       //single port output
185         .dpo(_2ndOp)                      //dual port output
186       );
187
188       ALU alu(
189        .i_is_add_op(i_is_add_op),
190        .i_a(_a),
191        .i_b(_b),
192        .i_carry_in(i_carry_in),
193        .i_logic_op_sel(i_logic_op_sel),
194        .i_shift_op_sel(i_shift_op_sel),
195        .o_arithmetic_result (_op_arithmetic_result),
196        .o_logic_result(_op_logic_result),
197        .o_shift_result(_op_shift_result),
198        .o_cc (o_cc)
199       );
200
201        /*Memory instantiation showed previously*/
202
203
204 endmodule
```

Listing 4.6: Control Unit instantiation

4.5 Timer and Parallel I/O

The code shown in Listing 4.7 presents the updated implementation of the timer peripheral and its associated control and status registers, as defined during the design phase.

The timer logic is implemented as a synchronous process driven by the rising edge of the clock. Upon reset, all control signals are cleared, the interrupt request output `o_int_req` is deasserted, and the counter initialization register `_cnt_init_reg` is loaded with its default value. The control register fields `_start`, `_timer`, and `_int_en` respectively enable the timer, select the timer mode, and control interrupt generation.

When the timer is active and a valid tick event occurs, an interrupt request is generated if interrupts are enabled. This behavior corresponds to the interrupt request register (CR#1), which asserts `o_int_req` and remains active until explicitly cleared by a write operation to the same register. This mechanism ensures a level-sensitive interrupt request that can be reliably acknowledged by the processor.

Register writes are decoded based on the address bus. Writing to CR#0 updates the timer control fields, writing to CR#1 clears the interrupt request, and writing to CR#2 updates the counter initialization value. This structure closely follows the memory-mapped peripheral model defined in the architectural specification.

```
1 always @(posedge _clk) begin
2     if (_rst) begin
3         {_start, _timer, _int_en} <= 3'b000;
4         o_int_req <= 1'b0;
5         _cnt_init_reg <= i_cnt_init;
6     end
7     else begin
```

```
8           //CR#1 interrupt request
9           if (_tick && _v && _int_en)
10          o_int_req <= 1'b1;
11
12          // Handle writes based on address
13          if (_we[0]) begin
14              case (_ad)
15                  // 0x1000 - Counter Control Register
16                  CR_0[7:0]:{_start, _timer, _int_en} <=
                        i_data_to_store[2:0];
17                  // 0x1002 - Interrupt Request Register
18                  CR_1[7:0]: o_int_req <= 1'b0;  // Clear on write
19                  // 0x1004 - Counter Init Register
20                  CR_2[7:0]:  _cnt_init_reg <= i_data_to_store;
21                  default: begin
22                  end
23              endcase
24          end
25      end
26  end
```

Listing 4.7: timer peripheral implemenatation

To validate the correct operation of the timer and its registers, a test program was executed. Initially, the counter was observed to start from `0x0000`. A write operation to CR#2 updated the counter initialization register to `0xFFE5`, as shown in Figure 4.3, confirming the correct behavior of the register write path.

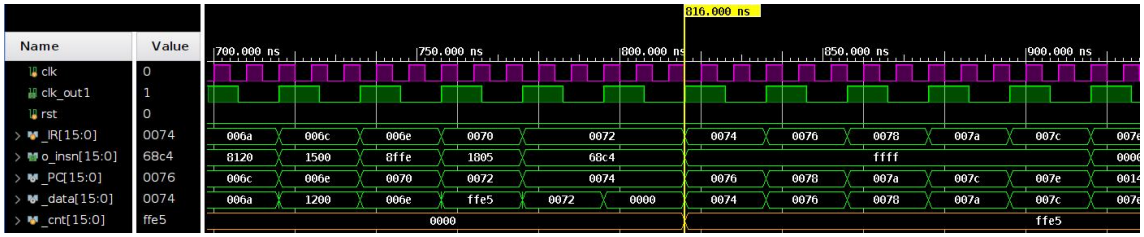

Figure 4.3: Reconfiguration of initial counter value

Further validation was performed by enabling the timer interrupt and executing an interrupt service routine designed during the design phase. After four timer overflows, bit 0 of the peripheral output register (`par_o`) toggles, demonstrating correct interrupt generation, handling, and interaction between the timer and the peripheral logic. This behavior is illustrated in Figure 4.4, where the LED output visibly toggles as expected.

These results confirm that the timer peripheral, its register interface, and the interrupt handling mechanism operate correctly and in accordance with the original design specifications.

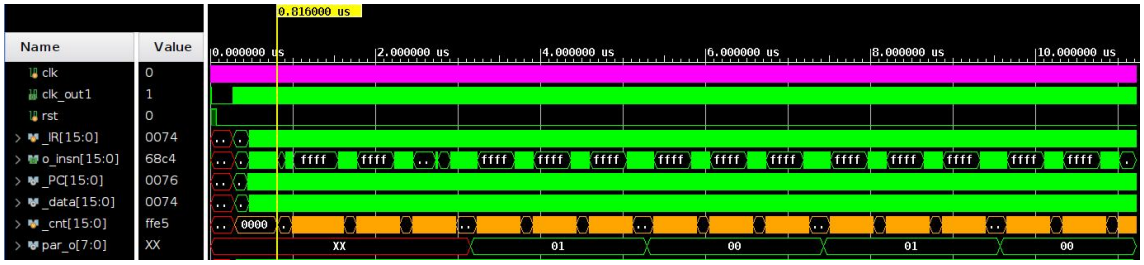

Figure 4.4: Test case using timer and parallel i/o

4.6 UART

As observed in the figure, the transmitted data on the `TX` line matches exactly the value received on the `RX` line. This behavior confirms the correct operation of both the UART receiver and transmitter, as well as the proper internal data handling and timing between the two modules. The successful echo of the received byte demonstrates that the UART subsystem functions as intended.

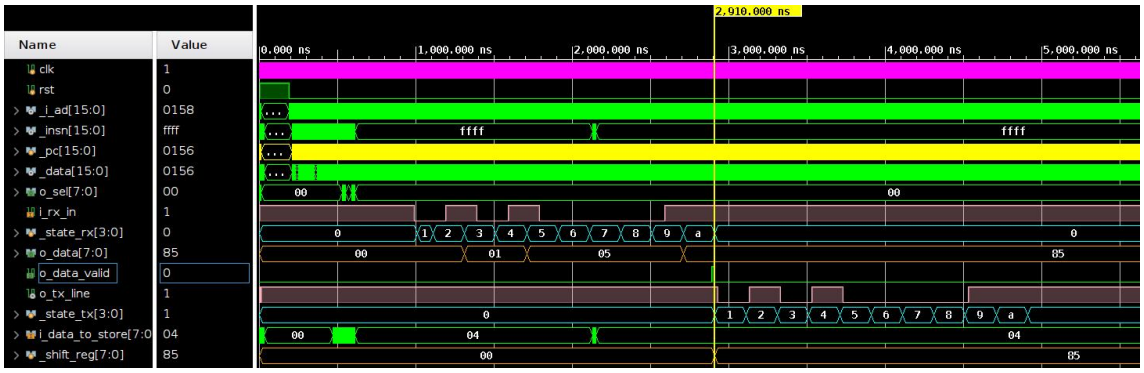

Figure 4.5: UART echo test case

Figure 4.5 shows a validation example of the UART peripheral configured in echo mode. In this test, a byte with value `0x0085` is transmitted on the UART receive (`RX`) line. After the reception is completed and the data is correctly decoded by the UART receiver, the same value is immediately forwarded to the UART transmit (`TX`) line.

```
always @(posedge _clk) begin
    if (~_rst) begin
        case (_state_tx)
            0: begin  // Idle
                o_tx_out <= 1;  // High when idle
                if (_tx_start) begin
                    _shift_reg <= _sbuf_tx;
                    _state_tx <= 1;
                    _counter <= 0;
                end
            end
            1: begin  // Start bit (0)
                o_tx_out <= 0;
                if (_counter == i_baudrate_div - 1) begin
                    _counter <= 0;
                    _state_tx <= 2;
                end else _counter <= _counter + 1;
            end
            2, 3, 4, 5, 6, 7, 8, 9: begin  // Data bits (LSB first)
                o_tx_out <= _shift_reg[_state_tx - 2];
                if (_counter == i_baudrate_div - 1) begin
                    _counter <= 0;
                    _state_tx <= _state_tx + 1;
                end else _counter <= _counter + 1;
            end
            10: begin  // Stop bit (1)
                o_tx_out <= 1;
                if (_counter == i_baudrate_div - 1) begin
                    _counter <= 0;
                    _state_tx <= 0;
                end else _counter <= _counter + 1;
            end
        endcase
```

```
    end
    else
        o_tx_out <= 1;
end
```

Listing 4.8: uart tx implemenatation

```
always @(posedge _clk) begin
    case (_state_rx )
        0: begin  // Wait for start bit (falling edge)
            if (!i_rx_in) begin
                _counter <= 0;
                _state_rx  <= 1;
            end
        end
        1: begin  // Sample mid-start bit
            if (_counter == _half_baudrate_div - 1) begin
                if (!i_rx_in) begin  // Confirm start bit
                    _counter <= 0;
                    _state_rx  <= 2;
                end else _state_rx  <= 0;  // False start
            end else _counter <= _counter + 1;
        end
        2, 3, 4, 5, 6, 7, 8, 9: begin  // Sample data bits
            if (_counter == i_baudrate_div - 1) begin
                _shift_reg[_state_rx  - 2] <= i_rx_in;  // Store
                   bit
                _counter <= 0;
                _state_rx  <= _state_rx  + 1;
            end else _counter <= _counter + 1;
        end
        10: begin  // Stop bit
            if (_counter == i_baudrate_div - 1) begin
                _counter <= 0;
                _state_rx  <= 0;
            end else _counter <= _counter + 1;
        end
    endcase
end
```

Listing 4.9: uart rx implemenatation

4.7 I2C

The I2C (Inter-Integrated Circuit) module implements a standard I2C protocol for communication with external peripherals. As shown in Figure 4.6, the waveform demonstrates successful I2C operation with proper START and STOP conditions, address transmission, and data transfer.

To initiate an I2C transaction, the processor must configure the I2C data register with the peripheral address in the least significant byte (LSB) and the data to be transmitted in the most significant byte (MSB). The I2C control register must have both the enable (EN) and start (START) flags set to 1 to begin the communication sequence.

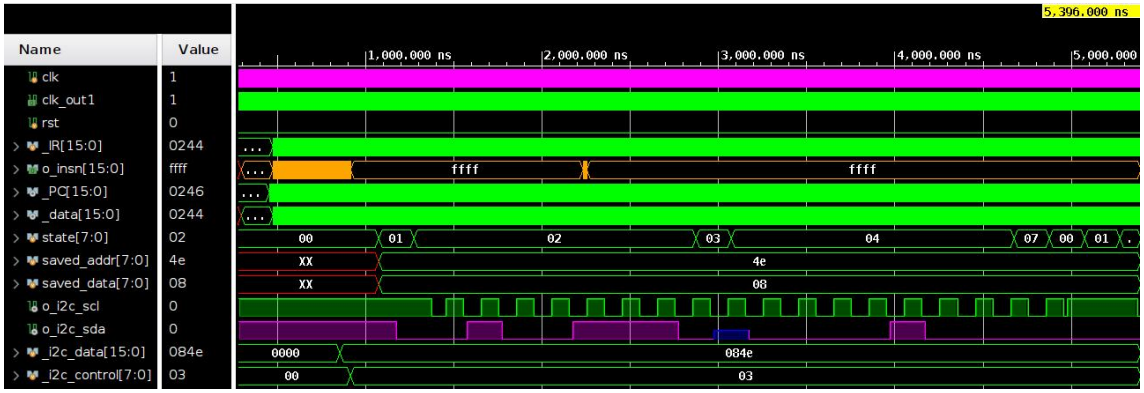

Figure 4.6: I2C Communication Waveform showing successful data transmission

The I2C implementation uses a dual-edge triggered state machine approach, where rising edges of the I2C clock handle state transitions and data sampling, while falling edges manage data output and the SDA line direction control. This ensures proper setup and hold times as required by the I2C specification.

```
// I2C posedge Validation
always @(posedge _clk) begin
    if(_rst == 1) begin
        state <= IDLE;
        counter <= 0;
        o_data <= 16'h0000;
    end
    else if (_i2c_control[bit_EN])begin
        if (i2c_clk_rising_en && clk_enable) begin
            case(state)

                IDLE: begin
                    if (_i2c_control[bit_START]) begin
                        state <= START;
                        saved_addr <= _i2c_data[7:0];
                        saved_data <= _i2c_data[15:8];
                    end
                    else state <= IDLE;
                end

                START: begin
                    counter <= `DATA_I2C-1;  // Set bit number for
                       transmission
                    state <= ADDRESS;
                end

                ADDRESS: begin
                    if (counter == 0) begin
                        state <= READ_ACK;
                    end else counter <= counter - 1;
                end

                READ_ACK: begin
                    if (o_i2c_sda == 0) begin
                        counter <= `DATA_I2C-1; // Set bit number
                           for transmission
                        if(saved_addr[0] == 0) state <= WRITE_DATA;
                        else state <= READ_DATA;
                    end else state <= STOP;
                end

```

```
                WRITE_DATA: begin
                    if(counter == 0) begin
                        state <= READ_ACK2;
                    end else counter <= counter - 1;
                end

                READ_ACK2: begin
                  if ((o_i2c_sda == 0) && (bit_START == 1)) state
                      <= IDLE;
                  else state <= STOP;
                  //state <= IDLE;
                end

                READ_DATA: begin
                    o_data[counter] <= o_i2c_sda;
                    if (counter == 0) state <= WRITE_ACK;
                    else counter <= counter - 1;
                end

                WRITE_ACK: begin
                    state <= STOP;
                end

                STOP: begin
                    state <= IDLE;
                end
            endcase
        end
    end
end
```

Listing 4.10: I2C Posedge State Machine - Handles state transitions and data reception

```
// I2C negedge Validation
always @(posedge _clk) begin
    if(_rst == 1) begin
        write_enable <= 1;
        sda_out <= 1;
    end else if (i2c_clk_falling_en && clk_enable) begin
        case(state)

            START: begin
                write_enable <= 1;
                sda_out <= 0;
            end

            ADDRESS: begin
                sda_out <= saved_addr[counter];
            end

            READ_ACK: begin
                write_enable <= 0;
            end

            WRITE_DATA: begin
                write_enable <= 1;
                sda_out <= saved_data[counter];
            end

            WRITE_ACK: begin
                write_enable <= 1;
```

```
                sda_out <= 0;
            end

            READ_DATA: begin
                write_enable <= 0;
            end

            STOP: begin
                write_enable <= 1;
                sda_out <= 1;
            end
        endcase
    end
end
```

Listing 4.11: I2C Negedge Logic - Controls SDA output and direction

The posedge state machine (Listing 4.10) manages the protocol sequencing, beginning in the IDLE state where it waits for the START condition to be asserted. Upon receiving the start signal, it captures the address and data from the I2C data register. The state machine then progresses through ADDRESS transmission, ACK reception, and either WRITE_DATA or READ_DATA states depending on the read/write bit (bit 0) of the address byte. Each state transition occurs on the rising edge of the I2C clock to properly sample incoming data.

The negedge logic (Listing 4.11) operates on falling edges of the I2C clock to set up data on the SDA line before the next rising edge. The `write_enable` signal controls the bidirectional nature of the SDA line, setting it to output mode during transmission and input mode during ACK and data reception phases. This dual-edge approach ensures compliance with I2C timing specifications, where data must be stable during the high period of SCL and can only change during the low period.

4.8 Interrupt Controller

Listing 4.12 shows the complete priority encoding implementation. The critical components of this implementation are explained below.

```
assign _int_served_mask = (1 << _int_index_next);

/* Assure instruction is interruptable */
wire _int_can_occur =  o_valid &                // Valid to insert
    (~rst and on a valid clk cycle (due to mem) - o_valid_insn_ce)
                    ~(`IMM|`ALU&(`ADC|`SBC|`CMP)) &  // Assure int
                       req only on NON interruptable instructions
                    ~_int_in_progress;              //Assure the
                       int being served is not interrupted

/* Logic to make sure the same interrupt isn't served twice,
    consecutively */
always @(posedge i_clk)
if (i_rst)
  _int_req_last <= 0;
else
  _int_req_last <= i_int_req;


assign _int_pend = (i_rst || (|o_int)) ? 0 : (i_int_req &
    ~_int_req_last);
wire _int_accept = (|_int_next) & o_insn_ce;

```

```
/* Interrupt in progress flag */
always @(posedge i_clk) begin
    if (i_rst)
        _int_in_progress <= 1'b0;
    else if (o_is_reti && o_valid_insn_ce)
        _int_in_progress <= 1'b0;
    else if (_int_accept)
        _int_in_progress <= 1'b1;
end

/* Pending interrupt queue (bitmask)*/
always @(posedge i_clk) begin
    if (i_rst)
        _int_waiting <= 0;
    else begin
        /* Add pending interrupt to queue */
        if (_int_in_progress && (|_int_pend))
            _int_waiting <= _int_waiting | _int_pend;

        /* Remove interrupt when it is accepted */
        if (_int_accept)
            _int_waiting <= _int_waiting & ~_int_served_mask;
    end
end

/* Next interrupt selection (priority encoder) */
   always @(posedge i_clk) begin
        if (i_rst)
            _int_next <= 0;
        else if (_int_accept)
            _int_next <= 0;
        else if (~_int_in_progress) begin
            if (|_int_waiting)
                _int_next <= _int_waiting;
            else if (|_int_pend)
                _int_next <= _int_pend;
        end
    end


/* Select Interrupt Request Line */
assign _int_index_next =   _int_next[0] ? 4'b0000:
                          (_int_next[1] ? 4'b0001 :
                          (_int_next[2] ? 4'b0010 :
                          (_int_next[3] ? 4'b0011 :
                          (_int_next[4] ? 4'b0100 : 4'b0000))));

/* Register the interrupt index when interrupt is accepted */
always @(posedge i_clk) begin
    if (i_rst)
        _int_index <= 4'b0000;
    else if (_int_accept)  // Capture when interrupt triggers
        _int_index <= _int_index_next;
end

/* Signal Interrupt to the System */
always @(posedge i_clk) begin
    if (i_rst)
        o_int <= `INT_REQ_LINES'b00000;
    else if (_int_accept)
        o_int <= _int_next;
```

```
80      else
81          o_int <= 'INT_REQ_LINES'b00000;
82  end
83
84  assign o_zero_insn = _int_accept;
85
86  /******************** HARDCODED INTERRUPT VECTOR TABLE
        ***********************/
87  assign o_i_ad_IVT  =  i_rst ? 16'h0002 :
88                        ((|o_int) ? {8'b0,_int_index, 1'b0, (|o_int),
                             2'b0} : 16'h0002);
```

Listing 4.12: Priority encoding and interrupt management logic

Critical Implementation Components

Interrupt Served Mask: The `_int_served_mask` signal creates a bitmask with a single bit set corresponding to the interrupt being serviced. This mask is crucial for removing the specific interrupt from the waiting queue using bitwise AND operation with the complement of the mask.

Interrupt Condition Check: The `_int_can_occur` signal enforces three critical conditions before allowing an interrupt:

1. The current instruction cycle must be valid
2. The current instruction must not be part of an interlocked sequence (IMM, ADC, SBC, CMP)
3. No interrupt service routine is currently executing (preventing nested interrupts)

Edge Detection Logic: The edge detection mechanism using `_int_req_last` prevents the same interrupt from being serviced multiple times. By comparing the current interrupt request with the previous cycle's value, only rising edges (new interrupt requests) are recognized as pending interrupts.

Interrupt In-Progress Flag: The `_int_in_progress` flag tracks the execution state of interrupt service routines. It is set when an interrupt is accepted and cleared only when a RETI instruction completes, ensuring that no new interrupts can interrupt an ongoing ISR.

Pending Interrupt Queue: The `_int_waiting` register maintains a bitmask of interrupts that arrived while an ISR was executing. When a new interrupt is detected during ISR execution, it is added to the queue using bitwise OR. When an interrupt is accepted for service, it is removed using bitwise AND with the complement of the served mask.

Priority Encoder: The priority selection logic in `_int_next` implements a simple but effective priority scheme: queued interrupts are serviced before newly pending ones, and within each category, lower-indexed interrupts have higher priority. This ensures fair servicing while maintaining deterministic priority levels.

Index Encoding: The `_int_index_next` combinational logic converts the one-hot encoded interrupt selection into a binary index. This cascaded ternary operator implements a priority encoder where interrupt 0 has the highest priority and interrupt 4 has the lowest.

Interrupt Vector Table Address Generation: The IVT address calculation follows a simple formula: each interrupt vector is located at `0x0002 + (interrupt_index × 4)`, providing 4 bytes of space per interrupt vector. This addressing scheme allows each interrupt to have a dedicated entry point in the vector table, with enough space for a short handler or a jump to the full ISR.

Validation

For testing this section, two interrupts were triggered while another interrupt was already in progress. As expected, the currently executing interrupt completes first, and then the higher-priority interrupt is serviced, even though it was activated after the lower-priority one.

```
‘timescale 1ns / 1ps

module tb_soc();

 reg clk, rst;
 reg [7:0] par_i;
 wire [7:0] par_o;

 SoC uut(.i_clk(clk), .i_rst(rst), .i_par_i(par_i),
    .o_par_o(par_o));

 initial clk = 0;
 always #10 clk = ~clk;

 initial begin
 rst = 1;  par_i = 0;

 repeat (10) @(posedge clk); rst = 0;

 repeat (14) @(posedge clk); par_i = 8'b01; // par_i[0] start timer
 repeat (1) @(posedge clk); par_i = 8'b00;
 repeat (38) @(posedge clk); par_i = 8'b10; // par_i[1]  stop timer
 repeat (2) @(posedge clk); par_i = 8'b0;
 repeat (2) @(posedge clk); par_i = 8'b01; // par_i[0] start timer
 repeat (1) @(posedge clk); par_i = 8'b00;

 repeat (40) @(posedge clk);
 $finish;

 end

endmodule
```

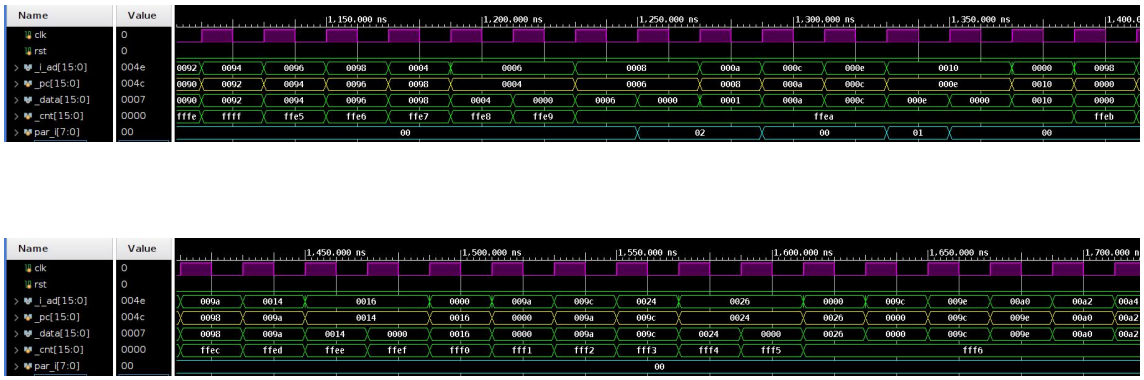

Figure 4.7: Priority Encoding test case

4.9 Latch Avoidance

4.9.1 Original Code

```
    reg br, t;
    always @(hit or cond or op or ccz or ccn or ccc or ccv) begin
      case (cond&4'b1110)
      `BR:   t = 1;
      `BEQ:  t = ccz;
      `BC:   t = ccc;
      `BV:   t = ccv;
      `BLT:  t = ccn^ccv;
      `BLE:  t = (ccn^ccv)|ccz;
      `BLTU: t = ~ccz&~ccc;
      `BLEU: t = ccz|~ccc;
      endcase
      br = hit & `Bx & (cond[0] ? ~t : t);
    end
```

4.9.2 Refactored Code

```
  reg _t;
  always @(o_valid or _cond or _op or
           _ccz or _ccn or _ccc or _ccv) begin
    case (_cond&4'b1110)
    `BR:   _t = 1;
    `BEQ:  _t = _ccz;
    `BC:   _t = _ccc;
    `BV:   _t = _ccv;
    `BLT:  _t = _ccn^_ccv;
    `BLE:  _t = (_ccn^_ccv)|_ccz;
    `BLTU: _t = ~_ccz&~_ccc;
    `BLEU: _t = _ccz|~_ccc;
    default: _t = 0;
    endcase
    o_br = o_valid & `Bx & (_cond[0] ? ~_t : _t);
  end
```

4.10 Removal of Internal Tri-State

4.10.1 Original Code

```
  assign data = sum_en      ? sum : 16'bz;
  assign data = (`ALU&`LOG) ? log : 16'bz;
  assign data = (`ALU&`SR)  ? sr  : 16'bz;
  assign data = `JAL        ? pc  : 16'bz;

  assign data[15:8] = w_oe ? di[15:8] : 8'bz;
  assign data[7:0]  = l_oe ? di[7:0]  : 8'bz;
  assign data[7:0]  = h_oe ? di[15:8] : 8'bz;
  assign data[15:8] = lb   ? 8'b0     : 8'bz;

 assign data[7:0]  = swsb ? do[7:0]  : 8'bz;
  assign data[15:8] = sb   ? do[7:0]  : 8'bz;
  assign data[15:8] = sw   ? do[15:8] : 8'bz;
```

4.10.2 Refactored Code and analysis

```
    assign _data =
    o_io_rdy ? _io_data :
    i_w_oe ? {_di[15:8], _di[7:0]} :
    i_l_oe ? {8'h00, _di[7:0]} :
    i_h_oe ? {8'h00, _di[15:8]} :

    (i_op_type_sel == 2'b00) ? _op_arithmetic_result :
    (i_op_type_sel == 2'b01) ? _op_logic_result      :
    (i_op_type_sel == 2'b10) ? _op_shift_result      :
    /*(i_op_type_sel == 2'b11)  */ _pc;
    // Default value is _pc, given the default value of
       o_op_type_sel in C.U.
```

4.11 System Validation and Testing

In this section, the final test case will be validated. We will present a step-by-step explanation, starting from the planning of the test case, through the testbench setup, and finally the different simulations and procedures required for complete verification.

4.11.1 Test Program

COE File and Instruction Memory

Coefficient (COE) files are text-based initialization files used to preload memory contents in FPGA designs. In Xilinx Vivado, COE files are commonly employed to initialize Block RAM (BRAM) or ROM modules. This ensures that memory elements contain predefined values immediately after FPGA configuration, eliminating the need for additional runtime loading logic.

In this project, COE files are used to initialize the instruction memory (ROM) of the processor. The ROM stores the complete program to be executed, with instructions fetched sequentially by the instruction fetch stage of the datapath. By embedding the instructions directly into the FPGA configuration, deterministic program execution from reset is guaranteed, simplifying both simulation and hardware validation.

The COE file defines the instruction words and their order in memory, starting from address zero. During operation, the program counter indexes this ROM, and the fetched instructions are decoded by the Control Unit and executed by the Datapath. This mechanism closely resembles the behavior of a fixed program memory in a microcontroller or embedded processor.

Instruction Program Description

```
1  JAL R0, R0, #0
2  BR #41
3  //ISR TIMER
4  SW r6, r12, #0
5  SW r10, r12, #2
6  CMP r15, r14
7  BEQ #124
8  ADD r14, r13
9  SW r7, r12, #0
10 RETI
11 FFFF
12 // ISR EXTERN 0
13 SW r7, r12, #0
14 IMM #084
15 ADDI r2, r0, #E
16 SW r2, r4, #2
17 ADDI r2, r0, #3
18 SW r2, r4, #4
19 RETI
20 FFFF
21 // ISR EXTERN 1
22 SW r6, r12, #0
23 IMM #004
24 ADDI r2, r0, #E
25 SW r2, r4, #2
26 ADDI r2, r0, #3
27 SW r2, r4, #4
28 RETI
29 FFFF
30 //ISR UART RX
31 LB r3, r5, #0
32 SB r3, r5, #1
33 LB r8, r5, #5
34 ADDI r3, r10, #2
35 AND r8, r3
36 SB r8, r5, #5
37 RETI
38 FFFF
39 //ISR UART TX
40 ADDI r8, r10, #2
41 XOR r9, r8
42 SB r9, r11, #2
43 LB r3, r5, #5
44 AND r3, r13
45 SB r3, r5, #5
46 RETI
47 FFFF
48 //MAIN PROGRAM
49 IMM #076
50 ADDI r15, r0, #9
51 ADDI r14, r0, #0
52 ADDI r13, r0, #1
53 IMM #100
54 ADDI r12, r0, #0
55 IMM #110
56 ADDI r11, r0, #0
57 ADDI r10, r0, #0
58 ADDI r7, r0, #7
```

```
59 ADDI r6, r0, #3
60 IMM #120
61 ADDI r5, r0, #0
62 IMM #130
63 ADDI r4, r0, #0
64 IMM #004
65 ADDI r2, r0, #E
66 SW r2, r4, #2
67 ADDI r2, r0, #3
68 SW r2, r4, #4
69 FFFF
70 FFFF
71 FFFF
72 FFFF
73 //...
74 //ISR TIMER CONTINUATION at 0x0100
75 BR #5
76 AND r14, r10
77 XOR r9, r13
78 SB r9, r11, #2
79 BR #-125
80 //FFFF
81 //...
```

The instruction program is organized around an interrupt-driven architecture, in accordance with the flowcharts presented in the system design section. Most of the system functionality is executed inside interrupt service routines (ISRs), while the main program is kept intentionally minimal.

Each interrupt routine corresponds directly to a specific event defined in the design flowcharts. External interrupts are triggered by user interaction through physical buttons, which initiate or stop system operations. The timer, UART reception, and UART transmission completion are all handled exclusively within their respective ISRs, ensuring fast response times and deterministic behavior. This structure allows the processor to remain idle or in a waiting state until an event occurs, closely matching real embedded system operation.

User interaction plays a key role in triggering program execution. The only functionality that remains in the main program is the initialization and execution of the I^2C communication.

This organization results in a clean separation between initialization code and event-driven execution, improves system responsiveness, and ensures that the implemented behavior follows precisely the logic defined in the design flowcharts.

Preloaded Register Usage To simplify program development and interrupt handling, a set of general-purpose registers is preloaded with constant values at initialization. This reduces the number of instructions required during normal execution, particularly inside interrupt service routines, where the instruction budget is limited.

By assigning fixed roles and predefined values to specific registers, frequently used constants such as peripheral base addresses, control values, and counters are always readily available. This approach eliminates repeated immediate loads, minimizes register pressure during interrupt execution, and results in clearer, more efficient, and deterministic code.

Table 4.1 summarizes the registers that are initialized with predefined values and their intended usage throughout the program.

Register	Initial Value	Purpose
$r15$	769	Auto-reload value for the timer (2-second period)
$r14$	0	Counter used to track timer overflows
$r13$	1	Constant value used for increments
$r12$	1000	Base address of the timer peripheral
$r11$	1100	Base address of the parallel I/O peripheral
$r10$	0	–
$r9$	–	Value written to `par_o` to toggle or control outputs
$r7$	7	Control value used to start the timer
$r6$	3	Control value used to stop the timer
$r5$	1200	Base address of the UART peripheral
$r4$	1300	Base address of the I2C peripheral

Table 4.1: Preloaded registers and their usage

Test Procedure and Objectives

Once the program is loaded, we aim to test the system with all peripherals and interrupts working simultaneously: the timer, UART, and I^2C, triggered by external interrupts.

The COE files include both the initial instructions and the interrupt service routines (ISRs) for handling external events. The test objectives are as follows:

- **Timer:** The timer should start and stop correctly based on its respective external interrupts, toggling a dedicated LED every 2 seconds.
- **UART:** Sending a character from the terminal triggers an interrupt. The ISR echoes the received character back to the terminal. Once transmission completes, a second interrupt toggles an LED to indicate the end of transmission.
- **I^2C:** At system startup, I^2C is used to initialize a LCD display. This occurs only during the initial setup phase.

This approach ensures that the processor, memory, and peripherals operate correctly in a fully integrated scenario, validating interrupt handling, concurrent peripheral operation, and system timing behavior.

4.11.2 Testbench Design

Before programming the FPGA, the design must be thoroughly verified through simulation to ensure that the processor, peripherals, and interrupt mechanisms behave correctly. Simulation allows potential design errors to be detected and corrected in a controlled environment, reducing the risk of issues when the system is implemented in hardware.

To perform this verification, a **testbench** is developed. A testbench is a separate module that emulates the external environment of the system under test. It provides inputs to the design, monitors outputs, and observes internal signals, enabling validation of system behavior under realistic conditions without requiring physical hardware. Through simulation, the testbench can model complex scenarios, including peripheral responses, data transmission, and external interrupts, allowing designers to verify that the design meets the intended functionality.

In our testbench in Listing 4.11.2, several key scenarios are simulated. First, the I^2C interface is tested by modeling a slave device that responds with an acknowledgment (ACK), verifying correct communication during the initial setup. Next, an external interrupt from `par_i` is simulated to start the timer, allowing observation of correct timer initialization and counting. The UART interface is also tested by simulating characters arriving on the RX line. This ensures that the UART interrupt is triggered correctly and that the received

data is properly echoed back. Finally, events are included to stop the timer, confirming that the timer can be halted and its associated interrupts are handled as expected.

```
`timescale 1ns / 1ps

module tb_soc();

 reg clk;
 reg rst;
 reg [7:0] par_i;
 wire [7:0] par_o;
 reg i_rx_line;
 reg [7:0] rx_message;
 wire o_tx_line;
 wire o_i2c_scl;
 wire o_i2c_sda;

 SoC uut(
    .i_clk(clk), .i_rst(rst), .i_par_i(par_i),
    .i_rx_line(i_rx_line), .o_par_o(par_o),
    .o_tx_line(o_tx_line), .o_i2c_scl(o_i2c_scl),
    .o_i2c_sda(o_i2c_sda));

 initial clk = 0;
 always #4 clk = ~clk;

 initial begin
 rst = 1;
 par_i = 0;

 rx_message = 8'h85;
 i_rx_line = 1'b1;

 repeat (10) @(posedge clk);
 rst = 0;
 //start timer
 repeat (100) @(posedge clk);
 par_i = 8'b01; // par_i[0]
 repeat (2.5) @(posedge clk);
 par_i = 8'b00;

 repeat(100) @(posedge clk);
  i_rx_line = 0;
 repeat (25) @(posedge clk);
 i_rx_line = rx_message[0];
 repeat (25) @(posedge clk);
 i_rx_line = rx_message[1];
 repeat (25) @(posedge clk);
 i_rx_line = rx_message[2];
 repeat (25) @(posedge clk);
 i_rx_line = rx_message[3];
 repeat (25) @(posedge clk);
 i_rx_line = rx_message[4];
 repeat (25) @(posedge clk);
 i_rx_line = rx_message[5];
 repeat (25) @(posedge clk);
 i_rx_line = rx_message[6];
 repeat (25) @(posedge clk);
 i_rx_line = rx_message[7];
 repeat (25) @(posedge clk);
 i_rx_line = 1;
```

```
59
60      //stop timer
61      repeat (95) @(posedge clk);
62      par_i = 8'b10; // par_i[1]
63      repeat (5) @(posedge clk);
64      par_i = 8'b0;
65
66      repeat (40) @(posedge clk);
67      $finish;
68
69      end
70
71  endmodule
```

4.11.3 Behavioral Simulation

The first stage of simulation performed is the behavioral simulation. Behavioral simulation verifies the logical functionality of the design without considering hardware-specific implementation details such as gate delays, routing, or timing constraints. It focuses solely on the correctness of the RTL (Register Transfer Level) code, ensuring that the design produces the expected outputs for given inputs. Behavioral simulation does not account for real propagation delays, clock skew, or setup and hold violations, but it is extremely useful for early validation of control logic, data paths, and peripheral interactions.

In our behavioral simulation, all major system functions were tested simultaneously. The figure shows the following key events:

- The timer starts and stops correctly (orange wave) in response to the `par_i` external interrupt. This confirms that interrupt handling and timer control logic operate as expected.
- The UART receives characters on the RX line (light blue wave). The data is immediately echoed back to the terminal (pink wave), and upon completion of the transmission, an external signal on `par_o` is toggled to indicate the end of the UART activity.
- The I^2C interface correctly acknowledges the slave device and successfully transmits the intended data, demonstrating that initialization and data transfer occur as intended(purple line).

Importantly, in behavioral simulation, no timing delays are present. Signals change instantaneously according to the RTL behavior, allowing us to verify the correctness of logical sequences and functional interactions without being affected by physical implementation constraints. This stage confirms that the processor, peripherals, and interrupts interact correctly under ideal conditions, forming a solid foundation for subsequent post-synthesis and timing simulations.

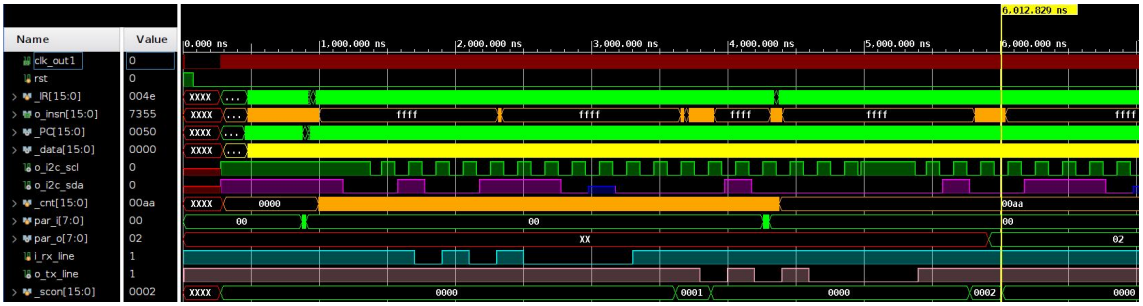

Figure 4.8: Behavioural Simulation of final test case

4.11.4 Post-Synthesis Timing Simulation and Synthesized Design

Before moving to post-synthesis simulation, certain RTL constructs were included to aid debugging and verification. For example:

```
(* keep = "true" *) reg [15:0] _cnt;
```

This synthesis attribute instructs the tool to preserve the register `_cnt` and prevents it from being optimized away, even if it appears unused in the logic. Using `(* keep = "true" *)` is particularly useful for waveform inspection, as it allows internal signals to remain visible in simulations and makes it possible to track values for timers, counters, or control signals during interrupt handling and peripheral operations.

Clock Wizard and Synchronous Reset

To ensure correct operation on the FPGA, the design includes a **clock wizard** and a synchronous reset mechanism. The clock wizard generates a stable 50 MHz clock from the input clock (`i_clk`) and provides a `locked` signal indicating when the output clock is stable. A synchronous reset, implemented with a 3-bit shift register, ensures that all registers and modules are properly initialized after the clock stabilizes, preventing metastability and ensuring deterministic reset release. The relevant code is shown below:

```
  wire _clk50;
  wire _locked;

  // Clock wizard instantiation
  clk_wiz_0 u_clk_wiz (
    .clk_in1 (i_clk),
    .clk_out1(_clk50),
    .reset   (i_rst),
    .locked  (_locked)
  );

  reg [2:0] _rst_sync;

  always @(posedge _clk50) begin
    if (i_rst || ~_locked)
      _rst_sync <= 3'b111;
    else
      _rst_sync <= {_rst_sync[1:0], 1'b0};
  end

  wire _rst_sys = _rst_sync[2];
```

Listing 4.13: Clock Wizard instantiation and synchronous reset

The shift register ensures that the system reset (`_rst_sys`) is deasserted only after three consecutive cycles of a stable clock. This guarantees that all synchronous modules—including the processor's Datapath, Control Unit, and peripheral buffers—are properly initialized.

Behavioral vs Post-Synthesis Simulation

Unlike Behavioural simulation, Post-synthesis timing simulation, includes actual gate-level delays introduced during synthesis, mapping, and placement on the FPGA. It takes into account routing delays, logic propagation, and timing constraints, providing a realistic view of how the design will behave on hardware. In our design, post-synthesis simulation confirmed that all interactions remain correct, with the only observable difference from behavioral simulation being a small propagation delay of approximately 18 ns. This minor delay does not affect the functional behavior of the timer, UART, I^2C, or interrupt handling.

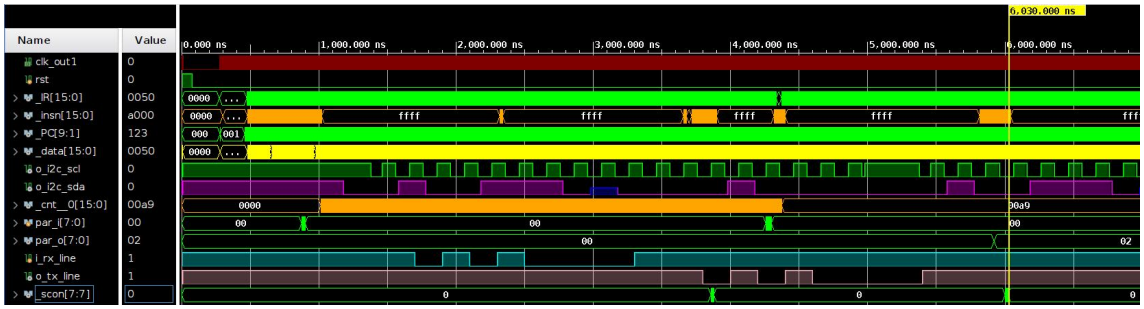

Figure 4.9: Post-Synthesis Timing Simulation

Gate-Level Design Outputs

After synthesis, the tool produces a gate-level netlist representing the actual logic that will be implemented on the FPGA. This synthesized design (Figure 4.10) clearly shows the processor's core components, including:

- The **Datapath**, responsible for arithmetic, logic, and data movement operations.
- The **Control Unit**, which generates the appropriate control signals to drive the Datapath and manage instruction execution.
- **Buffers and Registers for Parallel I/O**, which interface the processor with external peripherals while ensuring correct timing and synchronization.

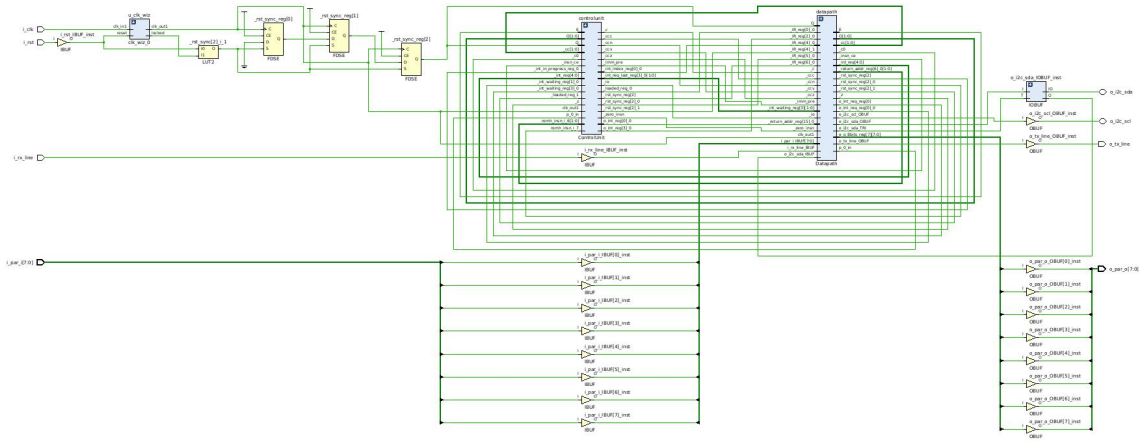

Figure 4.10: Synthesized Design

These outputs provide a detailed, hardware-accurate representation of the system, allowing for post-synthesis timing verification and ensuring that all components are correctly connected and optimized for FPGA implementation.

4.11.5 Post-Implementation Timing Simulation

Constraints

In FPGA design, constraints are essential directives that guide the synthesis and implementation tools to meet design requirements. Constraints are divided into two main categories: timing constraints and physical constraints. Understanding when and how to apply each type is crucial for achieving a functional and reliable design.

Timing constraints define the temporal requirements that the design must satisfy. They ensure that signals propagate through the logic within acceptable time windows, preventing setup and hold time violations. The primary timing constraints include:

- **Clock definitions**: Specify the clock frequency, duty cycle, and source using `create_clock`.
- **Input/Output delays**: Define the time relationship between external signals and the internal clock domain using `set_input_delay` and `set_output_delay`.
- **False paths**: Indicate signal paths that do not require timing analysis using `set_false_path`.
- **Multi-cycle paths**: Specify paths that intentionally take multiple clock cycles using `set_multicycle_path`.

Physical constraints control the placement and routing of design elements on the FPGA fabric. These constraints include:

- **Pin assignments**: Map design signals to specific FPGA package pins (LOC constraints).
- **I/O standards**: Define voltage levels and signaling standards (IOSTANDARD).
- **Area constraints**: Restrict logic placement to specific regions (AREA_GROUP, PBLOCK).
- **Placement constraints**: Fix the location of specific logic elements (BEL, RLOC).

Figure 4.11 shows the pin assignment constraints for the design, mapping signals to specific FPGA package pins with appropriate I/O standards (LVCMOS33 at 3.3V). The timing constraints applied are shown in Listing 4.14.

Name	Direction	Board Part Pin	Board Part Interface	Interface	Neg Diff Pair	Package Pin	Fixed	Bank	I/O Std	Vcco	Vref	Drive Strength
i_clk	IN					K17	☑	35	LVCMOS33*	3.300		
i_rst	IN					T16	☑	34	LVCMOS33*	3.300		
i_rx_line	IN					Y17	☑	34	LVCMOS33*	3.300		
o_tx_line	OUT					T17	☑	34	LVCMOS33*	3.300		12
i_par_i[7]	IN					U12	☑	34	LVCMOS33*	3.300		
i_par_i[6]	IN					T12	☑	34	LVCMOS33*	3.300		
i_par_i[5]	IN					Y14	☑	34	LVCMOS33*	3.300		
i_par_i[4]	IN					W14	☑	34	LVCMOS33*	3.300		
i_par_i[3]	IN					T10	☑	34	LVCMOS33*	3.300		
i_par_i[2]	IN					T11	☑	34	LVCMOS33*	3.300		
i_par_i[1]	IN					K19	☑	35	LVCMOS33*	3.300		
i_par_i[0]	IN					Y16	☑	34	LVCMOS33*	3.300		
o_par_o[7]	OUT					V18	☑	34	LVCMOS33*	3.300		12
o_par_o[6]	OUT					V17	☑	34	LVCMOS33*	3.300		12
o_par_o[5]	OUT					U15	☑	34	LVCMOS33*	3.300		12
o_par_o[4]	OUT					U14	☑	34	LVCMOS33*	3.300		12
o_par_o[3]	OUT					R14	☑	34	LVCMOS33*	3.300		12
o_par_o[2]	OUT					P14	☑	34	LVCMOS33*	3.300		12
o_par_o[1]	OUT					M15	☑	35	LVCMOS33*	3.300		12
o_par_o[0]	OUT					M14	☑	35	LVCMOS33*	3.300		12

Figure 4.11: Pin assignment constraints showing the mapping of design signals to FPGA package pins. Input signals (i_clk, i_rst, i_rx_line, i_par_i) and output signals (o_tx_line, o_par_o) are assigned to specific pins with LVCMOS33 I/O standard at 3.3V supply voltage.

```
# False path constraints for asynchronous inputs
set_false_path -from [get_ports {i_par_i[*]}]
set_false_path -from [get_ports i_rst]

# Input delay constraints (asynchronous, no relationship to clock)
set_input_delay -clock i_clk 0.000 [get_ports {i_par_i[*]}]
set_input_delay -clock i_clk 0.000 [get_ports i_rst]

# False path constraints for outputs (no external timing
    requirements)
set_false_path -to [get_ports {o_par_o[*]}]

# Output delay constraints
```

```
13 set_output_delay -clock i_clk 0.000 [get_ports {o_par_o[*]}]
```

Listing 4.14: Timing constraints file (XDC) for the design

The timing constraints define false paths for the parallel input bus (`i_par_i`) and reset signal (`i_rst`), as these are asynchronous signals not synchronized to the system clock. This prevents the timing analyzer from reporting false violations on these paths. Similarly, the parallel output bus (`o_par_o`) is marked as a false path since there are no external timing requirements for these signals—they are simply displayed on LEDs.

The zero input and output delays indicate that these signals do not have setup or hold time requirements relative to the clock edge, which is appropriate for asynchronous I/O and simple parallel interfaces. For high-speed synchronous interfaces, these values would need to reflect the actual timing requirements of the external devices.

Utilization Report

The utilization report, generated after place-and-route, summarizes how the design consumes FPGA resources. In this short example, shows the number and percentage of logic elements (LUTs, flip-flops), memory blocks (Block RAM), DSP slices, and I/O pins used by the implementation.

1. Slice Logic

Site Type	Used	Fixed	Prohibited	Available	Util%
Slice LUTs	469	0	0	17600	2.66
LUT as Logic	453	0	0	17600	2.57
LUT as Memory	16	0	0	6000	0.27
LUT as Distributed RAM	16	0			
LUT as Shift Register	0	0			
Slice Registers	331	0	0	35200	0.94
Register as Flip Flop	331	0	0	35200	0.94
Register as Latch	0	0	0	35200	0.00
F7 Muxes	2	0	0	8800	0.02
F8 Muxes	1	0	0	4400	0.02

3. Memory

Site Type	Used	Fixed	Prohibited	Available	Util%
Block RAM Tile	2	0	0	60	3.33
RAMB36/FIFO*	0	0	0	60	0.00
RAMB18	4	0	0	120	3.33
RAMB18E1 only	4				

* Note: Each Block RAM Tile only has one FIFO logic available and therefore can accommodate only one FIFO36E1 or one FIFO18E1. Tile, that tile can still accommodate a RAMB18E1

4. DSP

Site Type	Used	Fixed	Prohibited	Available	Util%
DSPs	0	0	0	80	0.00

5. IO and GT Specific

Site Type	Used	Fixed	Prohibited	Available	Util%
Bonded IOB	22	22	0	100	22.00
IOB Master Pads	12				
IOB Slave Pads	10				
Bonded IPADs	0	0	0	2	0.00
Bonded IOPADs	0	0	0	130	0.00
PHY_CONTROL	0	0	0	2	0.00
PHASER_REF	0	0	0	2	0.00

Figure 4.12: Utilization Report

This report is essential for verifying that the design fits within the target device and for assessing whether there is adequate resource headroom for future design modifications. It also helps determine if a smaller, more cost-effective FPGA could be used, or conversely, if a larger device is needed to accommodate additional functionality.

Timing Report

The timing report is a critical output generated during the implementation phase of the FPGA design flow, specifically after the place-and-route process has completed. Vivado's timing analysis engine performs static timing analysis (STA) to verify that all timing constraints specified in the design are met and that the circuit can operate reliably at the target clock frequency.

Static timing analysis examines all possible paths in the design to ensure that data can propagate from one register to another within the specified clock period. The analysis considers three main categories of timing checks:

- **Setup Time**: The minimum time before the clock edge that data must be stable at the input of a flip-flop
- **Hold Time**: The minimum time after the clock edge that data must remain stable at the input of a flip-flop
- **Pulse Width**: The minimum duration that a clock signal must remain high or low

Figure 4.13 presents the timing summary for the final implementation. This summary provides a high-level overview of the design's timing performance across all three categories of timing checks.

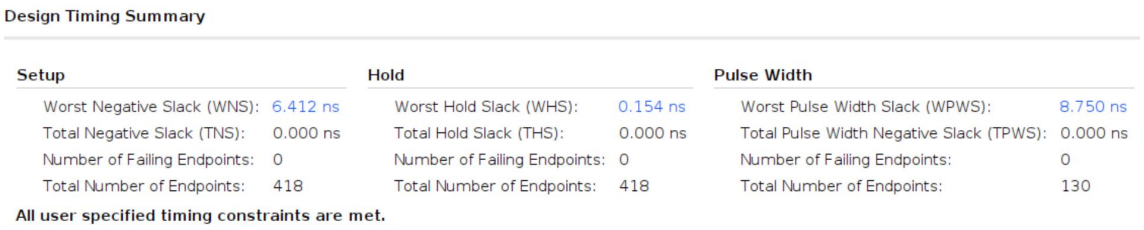

Figure 4.13: Timing summary showing setup, hold, and pulse width analysis results. All timing constraints are met with positive slack values.

Setup Timing Analysis: The setup timing analysis verifies that data arrives at destination registers with sufficient time before the clock edge:

- **Worst Negative Slack (WNS)**: 6.412 ns — This positive value indicates that the slowest path in the design has 6.412 ns of timing margin. This means the design can operate at a frequency higher than the specified 50 MHz (20 ns period). If this value were negative, it would indicate that at least one path is too slow and would fail to meet setup requirements by that amount.
- **Total Negative Slack (TNS)**: 0.000 ns — The sum of all setup violations across all endpoints. A value of zero confirms that no setup violations exist anywhere in the design.
- **Number of Failing Endpoints**: 0 — No paths fail to meet setup timing requirements.
- **Total Number of Endpoints**: 418 — The design contains 418 timing paths that were analyzed for setup timing.

Hold Timing Analysis: The hold timing analysis ensures that data remains stable for a sufficient duration after the clock edge:

- **Worst Hold Slack (WHS)**: 0.154 ns — The smallest hold margin in the design. This positive value confirms that all paths meet hold time requirements with at least 0.154 ns of margin.
- **Total Hold Slack (THS)**: 0.000 ns — The sum of all hold violations. Zero indicates no hold violations exist.
- **Number of Failing Endpoints**: 0 — All 418 endpoints meet hold timing requirements.

Pulse Width Analysis: The pulse width analysis verifies that clock signals maintain adequate high and low durations:

- **Worst Pulse Width Slack (WPWS)**: 8.750 ns — Significant margin exists for clock pulse width requirements.
- **Total Pulse Width Negative Slack (TPWS)**: 0.000 ns — No pulse width violations.
- **Number of Failing Endpoints**: 0 out of 130 clock pins — All clock signals meet pulse width requirements.

The summary conclusively states: **"All user specified timing constraints are met."** This confirms that the design is properly constrained and meets all timing requirements for reliable operation at the target frequency.

Figure 4.14 provides detailed information about the critical path—the slowest path in the design that determines the maximum operating frequency.

```
Max Delay Paths
--------------------------------------------------------------------------------------
Slack (MET) :             6.412ns  (required time - arrival time)
  Source:                 <hidden>
                            (rising edge-triggered cell RAMB18E1 clocked by clk  {rise@0.000ns fall@10.000ns period=20.000ns})
  Destination:            datapath/_pc_reg[12]/CE
                            (rising edge-triggered cell FDRE clocked by clk  {rise@0.000ns fall@10.000ns period=20.000ns})
  Path Group:             clk
  Path Type:              Setup (Max at Slow Process Corner)
  Requirement:            20.000ns  (clk rise@20.000ns - clk rise@0.000ns)
  Data Path Delay:        13.048ns  (logic 5.161ns (39.553%)  route 7.887ns (60.447%))
  Logic Levels:           10  (CARRY4=3 LUT2=1 LUT3=1 LUT4=1 LUT6=3 RAMD32=1)
  Clock Path Skew:        -0.097ns (DCD - SCD + CPR)
    Destination Clock Delay (DCD):    4.891ns = ( 24.892 - 20.000 )
    Source Clock Delay      (SCD):    5.380ns
    Clock Pessimism Removal (CPR):    0.391ns
  Clock Uncertainty:      0.035ns  ((TSJ^2 + TIJ^2)^1/2 + DJ) / 2 + PE
    Total System Jitter     (TSJ):    0.071ns
    Total Input Jitter      (TIJ):    0.000ns
    Discrete Jitter          (DJ):    0.000ns
    Phase Error              (PE):    0.000ns

    Location             Delay type                Incr(ns)  Path(ns)    Netlist Resource(s)
  -------------------------------------------------------------------    -------------------
                         (clock clk rise edge)        0.000     0.000 r
    K17                                               0.000     0.000 r  i_clk (IN)
                         net (fo=0)                   0.000     0.000    i_clk
    K17                                                         r  i_clk_IBUF_inst/I
    K17                  IBUF (Prop_ibuf_I_O)         1.492     1.492 r  i_clk_IBUF_inst/O
                         net (fo=1, routed)           2.076     3.568    i_clk_IBUF
    BUFGCTRL_X0Y16                                              r  i_clk_IBUF_BUFG_inst/I
    BUFGCTRL_X0Y16       BUFG (Prop_bufg_I_O)         0.101     3.669 r  i_clk_IBUF_BUFG_inst/O
                         net (fo=129, routed)         1.711     5.380    <hidden>
    RAMB18_X2Y12         RAMB18E1                                    r  <hidden>
  -------------------------------------------------------------------    -------------------
```

Figure 4.14: Detailed timing report showing the critical path breakdown, including source and destination registers, path delays, and contributing delay components.

The report also provides the physical location and netlist resources for each stage of the critical path, showing how the signal propagates from the Block RAM at location RAMB18_X2Y12 through various logic elements (LUTs, carry chains, buffers) until reaching the destination register. This level of detail is invaluable for understanding bottlenecks and optimizing critical paths if needed.

Pin Constraint Verification: Figure 4.15 shows the results of pin constraint verification, confirming that all I/O pins are properly constrained.

```
6. checking no_output_delay (0)
-------------------------------
 There are 0 ports with no output delay specified.

 There are 0 ports with no output delay but user has a false path constraint

 There are 0 ports with no output delay but with a timing clock defined on it or propagating through it
```

Figure 4.15: Pin constraint verification showing no unconstrained output pins or timing violations.

The verification report confirms:

- **No output delay (0)**: 0 ports with no output delay specified — all output pins have proper timing constraints
- **No false path constraints**: 0 ports with missing false path constraints — all paths that should be excluded from timing analysis are properly identified
- **No timing clock**: 0 ports with output delay but no timing clock defined — all constrained outputs have associated clock domains

This verification ensures that the pin constraints specified in the XDC (Xilinx Design Constraints) file are complete and correctly applied to all I/O ports. Proper I/O timing constraints are essential for ensuring that the FPGA can reliably interface with external devices.

The timing analysis results demonstrate that the design has been successfully implemented with significant timing margin.

The post-implementation timing report is the most accurate and is used to verify that the physical design meets all timing requirements. It accounts for actual wire delays, cell delays, and clock distribution delays that are only known after place-and-route completes. This report serves as the final sign-off that the design is ready for programming onto the FPGA and will operate correctly at the specified clock frequency.

Final Simulation

Figure 4.16 shows the post-implementation timing simulation of the complete system. This simulation incorporates all delays introduced by synthesis, placement, and routing on the target FPGA device, providing the most accurate representation of the design's timing behavior prior to actual hardware deployment.

Compared with post-synthesis timing simulation, the post-implementation simulation shows a slight recovery of approximately 11 ns due to optimization during placement and routing. Despite these minor timing adjustments, all functional objectives of the test case are fully met. The timer correctly starts and stops in response to external interrupts, UART data is received and echoed accurately, the UART transmission completion triggers the expected `par_o` toggle, and I^2C communication is correctly acknowledged and completed.

These results confirm that the design operates correctly under real hardware timing constraints and that the test case is ready for implementation on the FPGA.

Figure 4.16: Post-Implementation Timing Simulation

4.11.6 Hardware Validation

After completing simulation and post-implementation verification, the design was programmed onto the target FPGA for hardware validation. The test case was implemented using the following external connections:

- **External Interrupts:** Button1 (pin Y13) is connected to external interrupt 0 and is used to start the timer. Button2 (pin K19) is connected to external interrupt 1 and is used to stop the timer.
- **LED Indicators:** LED0 (pin M14) is configured as the blinking LED, driven by the timer with a 2-second period. LED1 (pin M15) toggles whenever a UART transmission completes, providing a visual indication of UART activity.
- **UART and I^2C Interfaces:** The FPGA UART is connected to an FTDI module. The FTDI RX pin is connected to the FPGA TX, and the FTDI TX is connected to the FPGA RX. Terminal communication is established via a serial terminal (e.g., using the command `sudo screen /dev/ttyUSB0 115200`) to send characters and observe the echo response. Additionally, the I^2C lines (SDA and SCL) are connected to the corresponding pins on the peripheral board for initialization and data transfer.

During validation, the following behavior was observed: the timer started and stopped correctly in response to the external interrupts, LED0 blinked with the expected 2-second period, and LED1 toggled after each UART transmission completion. Characters sent through the terminal were echoed back accurately via the FPGA UART, demonstrating proper interrupt handling and peripheral interaction. The I^2C initialization also completed successfully, confirming correct communication with the connected devices.

The following photographs illustrate the FPGA setup, button connections, LEDs, and terminal interaction during hardware testing:

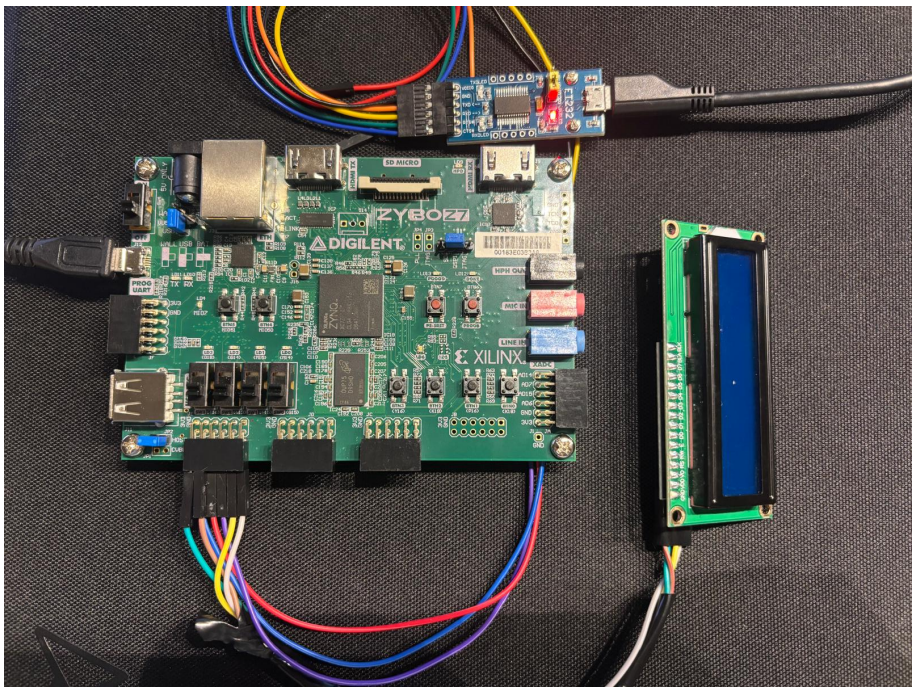

Figure 4.17: FPGA before configuration

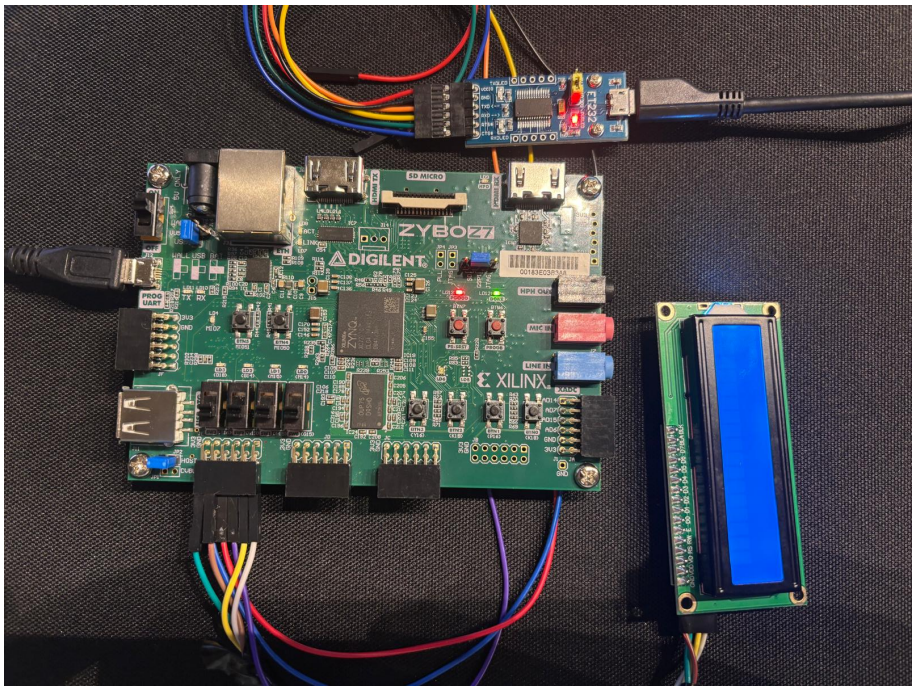

Figure 4.18: Program loaded in the FPGA and main finished: display turned on and no interrupts happened

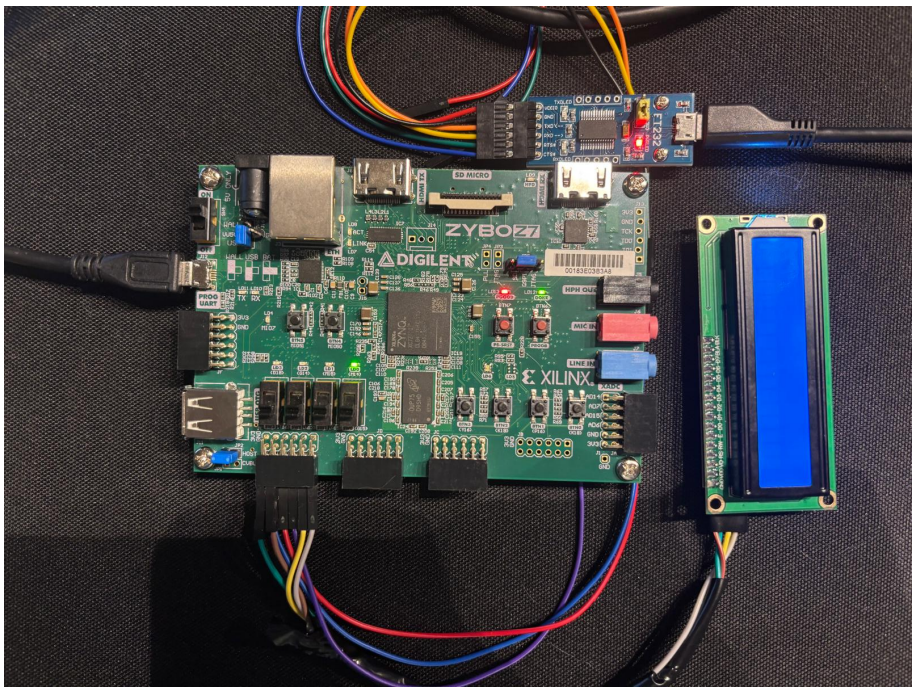

Figure 4.19: Button 1 pressed, timer started running and led 0 toggled

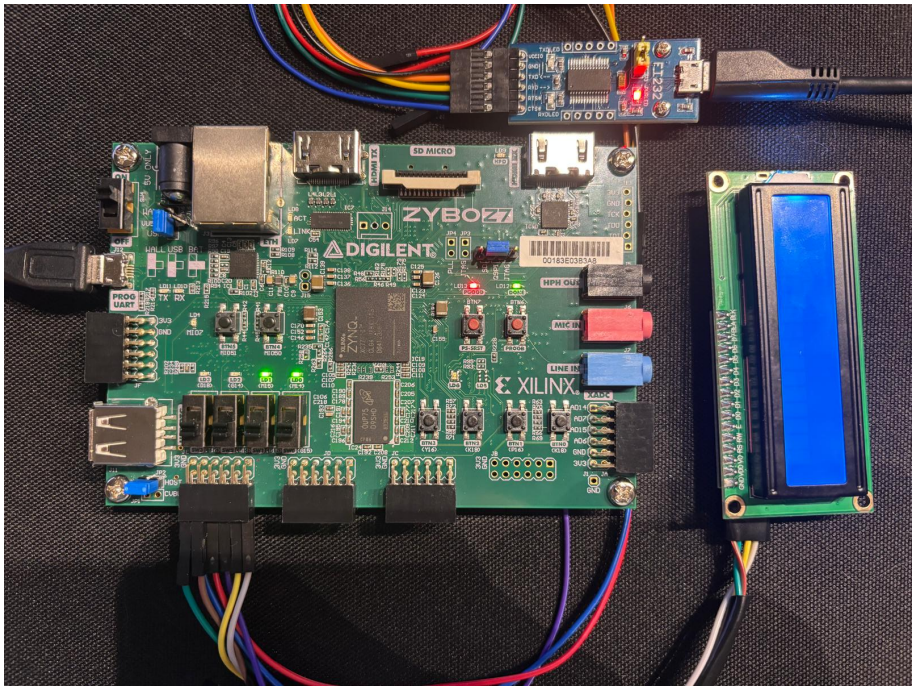

Figure 4.20: Button 2 pressed, timer stop running and led 0 stayed in the same status; Character was received and echoed so led 1 toogled

Figure 4.21: Echoed character from FPGA

Conclusion

This project successfully refactored and enhanced the GR0040 System-on-Chip while preserving full functional equivalence with the original design. The redesigned architecture improves modularity, maintainability, and extensibility through a clear separation of datapath and control logic, a cleaner memory organization, and standardized HDL practices.

Key enhancements include a slightly expanded instruction set, a robust priority-based interrupt controller, and fully functional peripherals such as a timer, UART, and I2C interface, with SFRs specification. Comprehensive simulation, timing analysis, and hardware validation on the Zybo Z7-10 FPGA confirmed correct operation.

Beyond meeting all technical objectives, the project provided valuable hands-on experience in processor design, Verilog HDL, FPGA implementation, and system-level debugging. Undoubtedly, the resulting SoC forms a solid foundation for future extensions.

Bibliography

1. http://www.fpgacpu.org/papers/soc-gr0040-paper.pdf
2. https://digilent.com/reference/programmable-logic/zybo-z7/reference-manual?srsltid=AfmBOoqTFpjJn9WPr3BxTcB_cOGEDRAdSx-N_vSIQe6iw_-96VB-ttbH
3. https://docs.amd.com/