

Technische Universität Dresden

Faculty of Electrical Engineering and Information Technology

Chair of Highly-Parallel VLSI Systems and Neuro-Microelectronics

VLSI Processor Design

Design of Dobby RISC-V SoC

Submitted by

PRZ22/Group 13

Janakiram Pandi (5038787) [paja22]

Keerthan Koppam Radhakrishna (5039224) [koke22]

Nitin Krishna Venkatesan (5039262) [veni22]

Rohan Krishna Vijayaraghavan (5040414) [viro22]

Date of Submission: 15 September 2023

Degree Program: M.Sc in Nanoelectronic Systems

Year of Matriculation: 2021

Supervisor: Dipl.-Ing. Stefan Scholze

Professor: Prof. Dr.-Ing. habil. Christian Mayr

Selbständigkeitserklärung

Statement of Authorship

We hereby declare that we have written this project work independently and have listed all used sources and aids. We are submitting this project work for the first time as part of an examination. We understand that attempted deceit will result in the failing grade „not sufficient“ (5.0).

Name / Last Name: Pandi

Vorname / First Name: Janakiram

Matrikelnummer / Matriculation Number: 5038787

15.09.2023, Dresden

Datum / Date



Unterschrift / Signature

Name / Last Name: Koppam Radhakrishna

Vorname / First Name: Keerthan

Matrikelnummer / Matriculation Number: 5039224

15.09.2023, Dresden

Datum / Date



Unterschrift / Signature

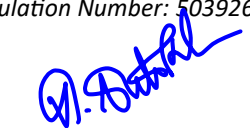
Name / Last Name: Venkatesan

Vorname / First Name: Nitin Krishna

Matrikelnummer / Matriculation Number: 5039262

15.09.2023, Dresden

Datum / Date



Unterschrift / Signature

Name / Last Name: Vijayaraghavan

Vorname / First Name: Rohan Krishna

Matrikelnummer / Matriculation Number: 5040414

15.09.2023, Dresden

Datum / Date



Unterschrift / Signature

Segregation of Work

Member	RTL Development / Sanity verification / Physical implementation	Testing/coverage verification
Janakiram Pandi	Synthesis Physical Design	Coverage for PRAM, Memory controller, init_controller and ALU – Dobby SoC simulation
Keerthan Koppam Radhakrishna	Fetch unit - RV32IM decoder - Control unit - ALU - RISC-V top – Register file	Coverage for fetch unit, RV32IM decoder, control unit, RISC-V top, and Register File
Nitin Krishna Venkatesan	Compression decoder - Memory controller - Trim unit - Load/store controller - Initialization controller - PRAM - Bus controller - Pads - Top Memory Controller	Memory initialization - Coverage for bus controller, init_controller - External I/O bus testing with external memory
Rohan Krishna Vijayaraghavan	CSR register file - PRAM - RISC-V top unit - PC unit - External memory peripheral	Verification of Dobby SoC - Self-checking testbench including External Bus interface - Generation of testcases in ASM & C - Sanity testcases for each instruction type

Contents

List of Figures	III
List of Tables	V
1 Introduction	1
1.1 Task description	1
1.2 Tools and Methodologies	2
1.3 Introduction to RISC-V ISA and its Extensions (RV32IMC)	3
1.3.1 RV32I Extension	5
1.3.2 RV32M Extension	7
1.3.3 RV32C Extension	7
2 Design and Implementation	10
2.1 RISC-V Core	10
2.1.1 Fetch Unit	11
2.1.2 PC Unit	11
2.1.3 Decoder	13
2.1.4 Compression Decoder	14
2.1.5 ALU	14
2.1.6 Register file	15
2.1.7 Control Status Registers	16
2.2 Top Memory Controller	17
2.2.1 Memory Controller	18
2.2.2 Program Memory	18
2.2.3 Bus Controller	20
2.2.4 Initialization Controller	21
2.3 Data and control paths of the SoC	22
2.4 Design optimizations to meet the design goals	23
2.4.1 Single Cycle Design	23
2.4.2 Two-stage Design	23
2.4.3 Breaking down the critical path	24
2.4.4 Optimizing the critical path	24
2.4.5 Memory optimization	24
2.4.6 Delay for each Instruction type	24
3 Simulation and Verification	26
3.1 Testbench	26

3.2	Simulation Setup	27
3.3	Verification Test cases	27
3.3.1	RV32IM ISA with ASM code	27
3.3.2	RV32IMC ISA with C codes	34
4	Synthesis and Physical Design	38
4.1	Synthesizability check of Dobby SoC	38
4.2	Synthesis Results	39
4.3	Physical Design	44
4.3.1	Bind	45
4.3.2	Floor Planning and Power Planning	45
4.3.3	Placement	46
4.3.4	Clock Tree Synthesis	46
4.3.5	Routing	46
4.3.6	Signoff	48
5	Conclusion	49
6	References	50
A	Appendix	51

List of Figures

1.1	Block diagram of Dobby SoC	2
1.2	RV32I instruction set	5
1.3	Supported instructions under RISC-V 'I' Extension	6
1.4	RV32M instruction set	7
1.5	RV32C instruction set	8
2.1	Block diagram of Dobby SoC	10
2.2	Block diagram of Dobby Core	11
2.3	Block diagram of the PC Unit	12
2.4	Coverage report for the PC unit	12
2.5	Block diagram of the Decoder unit	13
2.6	Coverage report for the Decoder unit	14
2.7	Coverage report for the Compression decoder unit	14
2.8	Block diagram of the ALU	15
2.9	Coverage report for the ALU unit	15
2.10	Block diagram of the Regfile	16
2.11	Coverage report for the reg file unit	16
2.12	Block diagram of the Memory Controller	18
2.13	Coverage report for the memory controller unit	18
2.14	Block diagram of the Program Memory	19
2.15	Coverage report for the PRAM unit	20
2.16	Block diagram of the bus controller unit	20
2.17	Coverage report for the bus controller unit	21
2.18	FSM in the initialization controller	22
2.19	Coverage report for the initialization controller	22
2.20	Datapath of the Core	23
3.1	Overview of Testbench Environment	26
3.2	PRAM loading	28
3.3	Regfile loading	28
3.4	Loading from external memory	29
3.5	Loading byte, word, half-word into regfile	29
3.6	Loading words from PRAM	30
3.7	Series of single cycle instructions	30
3.8	PC updation in Jump instructions	31
3.9	Store operations to PRAM and EXT	32
3.10	CSR Atomic instructions	33

3.11 Waveforms of System instructions	33
3.12 Requesting an Interrupt during WFI	34
3.13 factorial.c	35
3.14 bubble.c	35
3.15 armstrong.c	36
3.16 matmul.c	37
4.1 Depiction of HAL check log for the module doobby_soc	38
4.2 Synthesis constraints specified in the constraints.tcl file	39
4.3 Slack (REG2REG)	40
4.4 Critical Path (REGIN)	41
4.5 Critical Path (REGOUT)	42
4.6 Power consumption report	43
4.7 Area report	43
4.8 Dobby SoC schematic after synthesis	44
4.9 Flowchart of steps involved in the physical design phase	45
4.10 PostRoute setup timing Histogram	47
4.11 PostRoute Hold timing Histogram	47
4.12 Chip layout	48
A.1 Bus controller handling wait cycles during a bus write operation	51
A.2 Bus controller handling wait cycles during a bus read operation	51

List of Tables

1.1	RV32 instruction format	4
1.2	Supported instructions under RISC-V 'M' Extension	7
1.3	Supported compressed instructions and their equivalent RV32I instruction	9
2.1	PRAM Address Mapping to SRAM Macros	19
2.2	Total clock cycles for all instruction types	25

1 Introduction

In recent years, the computer architecture landscape has witnessed a transformative shift with the arrival of [Reduced Instruction Set Computer - V \(RISC-V\)](#), an open-source [Instruction Set Architecture \(ISA\)](#) standard. This evolution is not only about offering an alternative to conventional architectures; it's about rethinking the very paradigms of chip design. This is not just a change in nomenclature, but a comprehensive rethinking of the principles of chip design, which is supported by several technical advantages.

The basis of RISC-V's appeal is its simplicity and efficiency, embedded in a reduced instruction set. Unlike its complex counterparts, RISC-V focuses on fewer core instructions, resulting in more efficient, power-efficient, and faster execution. This design ethos makes it a compelling choice for a wide range of applications, from small embedded systems to high-performance computing environments.

Additionally, RISC-V's modularity is a game changer. Its extensibility allows designers to incorporate only the necessary components for their particular application, eliminating the unnecessary part of the core specification. This modularity also means that it can be adapted and scaled relatively easily to meet both current requirements and future technology trends.

Last, but by no means least, is the open-source nature of the RISC-V standard. Far from being just a philosophical stance, this transparency means tangible technical benefits. It enables a collaborative approach to problem-solving, rapid bug detection and resolution, and a constant stream of community-driven innovation.

1.1 Task description

The primary goal of this project is to design and implement an HPSN RISC-V microcontroller named "Dobby". Based on the RISC-V ISA, this is designed to be integrated into a [System On-Chip \(SoC\)](#) for general processing tasks. The main features and requirements for Dobby include:

- RISC-V ISA with IMC Extensions.
- A fully synchronous single-clock architecture.
- 16 kB of built-in [Program Memory \(PRAM\)](#).
- 2 general purpose interrupts

- A high-speed bus interface tailored for peripheral devices and external memory connections.
- An initiation mechanism for PRAM post-reset using the initialization controller.
- GCC compatibility for higher-level software development.
- Incorporation of an external bus master and memory controller, granting both the processor core and initialization controller access to the on-chip memory and external bus interface.

The [Register Transfer Level \(RTL\)](#) implementation of this SoC will be synthesized as a part of this project work. After synthesis, the Place and Route phases and the generation of the [Geometrical Database Standard for Information Interchange \(GDSII\)](#) file are performed. In addition to these design tasks, an optimization goal must be selected from power, performance, or area. For the scope of this project, the chosen goal of optimization is **performance**. The design will be analyzed and optimized to maximize operational speed and efficiency. [Figure 1.1](#) presents the architectural block diagram of the Dobby SoC.

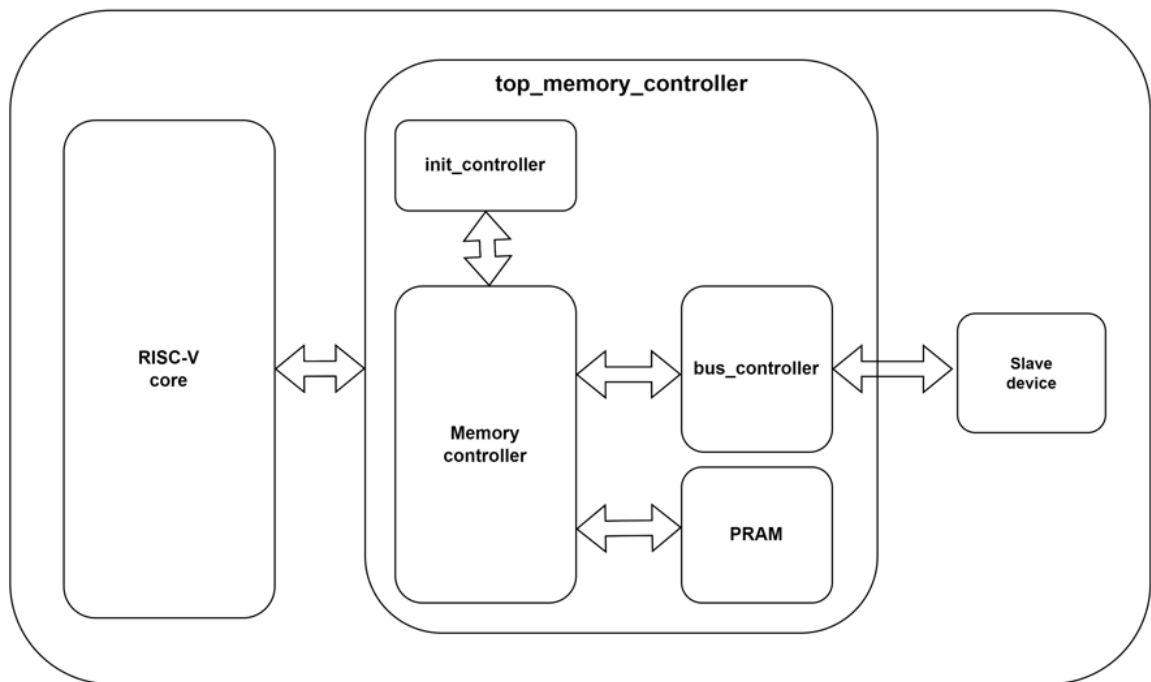


Figure 1.1: Block diagram of Dobby SoC

1.2 Tools and Methodologies

This project uses various industry standard tools and design methodologies for the entire [RTL-to-GDSII \(RTL2GDS\)](#) implementation flow. The [lab-icdesign.et.tu-dresden.de](#) computing servers with `eesl1` and `eesl2` machines are accessed using a [Virtual Network Computing \(VNC\)](#) client that allows remote access and development. The ICPRO Project and Design Flow

Management System (Version 1.0) is used to manage design resources according to a strict unit-based directory hierarchy. Each unit contains directories for the Verilog source code along with additional directories for the design steps. Compliance with dedicated design guidelines is checked using the **Cadence HAL RTL Checker tool** [7], which enables easy reuse of design blocks and team collaboration.

For RTL simulation and coverage analysis, **Cadence NCSIM** [8] is widely used both in the initial RTL phase, netlist phase, and post layout. **Synopsys Design Compiler** [9], the industry standard for RTL synthesis, is used to synthesize RTL and generate an optimized gate-level netlist. SRAM-based program memory, a key component, is initialized with a file to exactly emulate the desired memory function. **Cadence Innovus Implementation System** [10] is used for floor planning, placement, clock tree synthesis, and routing to create the final layout. TCL scripts are used to ensure reproducible results at each stage of the flow.

This comprehensive tool flow provides a structured implementation process from RTL to layout, leveraging established [Electronic Design Automation \(EDA\)](#) tools at each stage. The methodologies and tools used are consistent with standard [Application Specific Integrated Circuit \(ASIC\)](#) design practices, enabling the efficient design and integration of a RISC-V processor system on a chip.

1.3 Introduction to RISC-V ISA and its Extensions (RV32IMC)

RISC-V is known for its modularity in the world of instruction set architectures, and this is prominently showcased in its various instruction extensions. It offers a base instruction set that can be expanded with various extensions, each tailored for specific functionalities. Prominent among these extensions are the standard "M" for integer multiplication and division, "A" for atomic operations ensuring multi-processor synchronization, "F" and "D" for single and double-precision floating-point operations respectively, and "C" which introduces compressed 16-bit instructions to enhance code density. Additionally, there are more specialized extensions like "V" for vector operations and "B" for bit manipulation.

In RISC-V, instructions are classified into different types based on their structure and function, with the main types being:

- R-type - Register to Register
- I-type - Short immediates and loads
- J-type - Unconditional Jumps
- S-type - Stores
- B-type - Conditional branches
- U-type - Long immediates

Each type, defined by its unique format, serves specific computational tasks, from register operations to memory access and control transfers.

Every instruction type in RISC-V, has a 7-bit opcode, as indicated in [Table 1.1](#). The R-type instructions, in particular, are equipped with three distinctive register fields: rs1, rs2, and rd, corresponding to two source registers and one destination register. Each immediate field is labelled based on the bit position in the produced immediate value, used in decoding.

31	30	25	24	21	20	19	15	14	12	11	8	7	6	0			
funct7				rs2			rs1		funct3		rd		opcode		R-type		
imm[11:0]						rs1		funct3		rd		opcode		I-type			
imm[11:5]				rs2			rs1		funct3		imm[4:0]		opcode		S-type		
imm[12]		imm[10:5]			rs2			rs1		funct3		imm[4:1]		imm[11]		opcode	B-type
imm[31:12]										rd				opcode		U-type	
imm[20]		imm[10:1]			imm[11]		imm[19:12]				rd		opcode		J-type		

Table 1.1: RV32 instruction format

I-type instructions, as shown in [Table 1.1](#) [6], use two arguments, rs and rd, paired with a 12-bit immediate value which is a part of the instruction, offsetting the need for memory storage. This design principle improves processing speed, as accessing constants (immediate values) becomes fast without memory access. For control transfer, RISC-V has both unconditional (J-type) and conditional branches (B-type). The J-type format serves the unconditional jumps, like JAL, while all conditional branch instructions adopt the B-type format, with its 12-bit B-immediate efficiently encoding signed offsets.

This aids in quickly deciding the target address by adding to the current [Program Counter \(PC\)](#). Further, load and store instructions, necessary for data exchange between CPU registers and memory, use the I-type and S-type formats respectively. Lastly, the U-type instruction reflects the I-type in many ways but encodes only the destination register, using the upper immediate value for computational tasks.

1.3.1 RV32I Extension

The RV32I extension, representing the Integer base extension of RISC-V, is the foundation from which the architecture derives its flexibility and modularity. This extension primarily caters to operations on integer data types. Figure 1.2 [6] and Figure 1.3 [6] shows the RV32I instruction set with all the supported instructions.

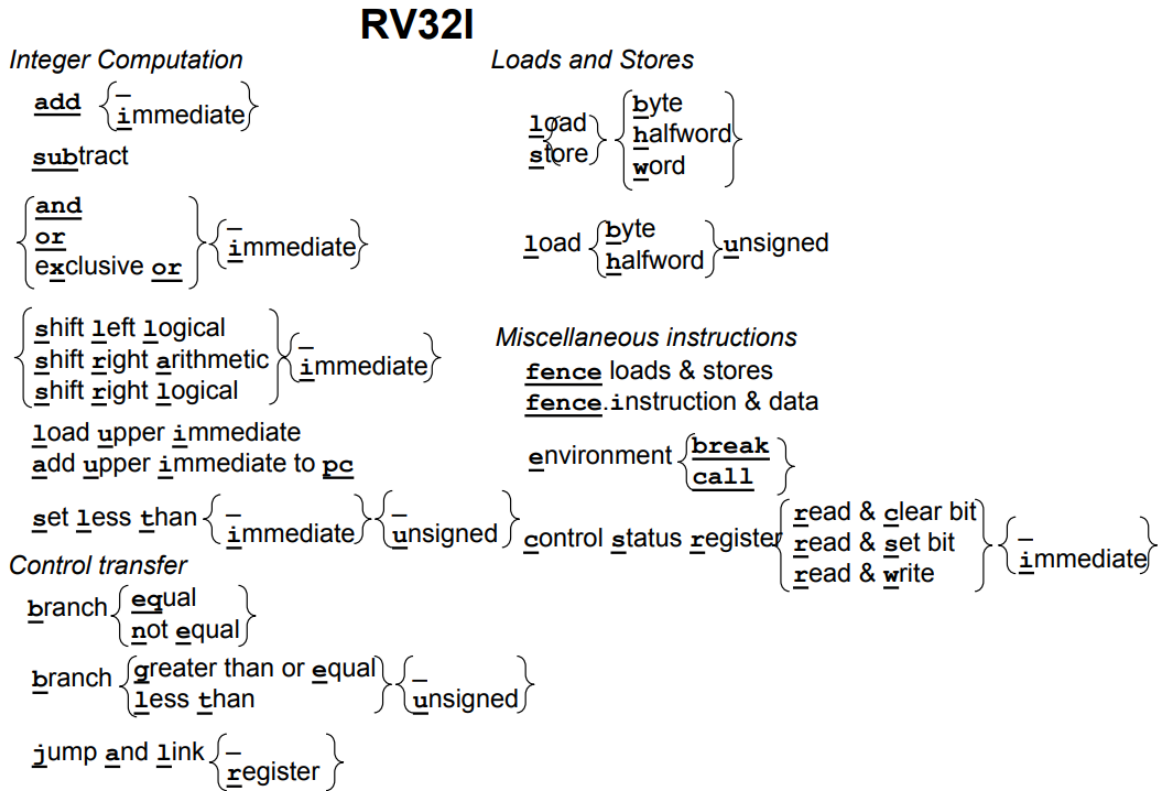


Figure 1.2: RV32I instruction set

Arithmetic instructions within the I extension cover both basic operations like addition (ADD and ADDI), subtraction (SUB) and bitwise operations (AND, OR, XOR, and their immediate counterparts). The shift operations (SLL, SRL, and SRA) are used for bit manipulation in the control flow of the program.

Base Integer Instructions: RV32I, RV64I, and RV128I				
Category	Name	Fmt	RV32I Base	+RV{64,128}
Loads	Load Byte	I	LB rd,rs1,imm	
	Load Halfword	I	LH rd,rs1,imm	
	Load Word	I	LW rd,rs1,imm	L{D Q} rd,rs1,imm
	Load Byte Unsigned	I	LBU rd,rs1,imm	
	Load Half Unsigned	I	LHU rd,rs1,imm	L{W D}U rd,rs1,imm
Stores	Store Byte	S	SB rs1,rs2,imm	
	Store Halfword	S	SH rs1,rs2,imm	
	Store Word	S	SW rs1,rs2,imm	S{D Q} rs1,rs2,imm
Shifts	Shift Left	R	SLL rd,rs1,rs2	SLL{W D} rd,rs1,rs2
	Shift Left Immediate	I	SLLI rd,rs1,shamt	SLLI{W D} rd,rs1,shamt
	Shift Right	R	SRL rd,rs1,rs2	SRL{W D} rd,rs1,rs2
	Shift Right Immediate	I	SRLI rd,rs1,shamt	SRLI{W D} rd,rs1,shamt
	Shift Right Arithmetic	R	SRA rd,rs1,rs2	SRA{W D} rd,rs1,rs2
	Shift Right Arith Imm	I	SRAI rd,rs1,shamt	SRAI{W D} rd,rs1,shamt
Arithmetic	ADD	R	ADD rd,rs1,rs2	ADD{W D} rd,rs1,rs2
	ADD Immediate	I	ADDI rd,rs1,imm	ADDI{W D} rd,rs1,imm
	SUBtract	R	SUB rd,rs1,rs2	SUB{W D} rd,rs1,rs2
	Load Upper Imm	U	LUI rd,imm	
	Add Upper Imm to PC	U	AUIPC rd,imm	
Logical	XOR	R	XOR rd,rs1,rs2	
	XOR Immediate	I	XORI rd,rs1,imm	
	OR	R	OR rd,rs1,rs2	
	OR Immediate	I	ORI rd,rs1,imm	
	AND	R	AND rd,rs1,rs2	
	AND Immediate	I	ANDI rd,rs1,imm	
Compare	Set <	R	SLT rd,rs1,rs2	
	Set < Immediate	I	SLTI rd,rs1,imm	
	Set < Unsigned	R	SLTU rd,rs1,rs2	
	Set < Imm Unsigned	I	SLTIU rd,rs1,imm	
Branches	Branch =	SB	BEQ rs1,rs2,imm	
	Branch ≠	SB	BNE rs1,rs2,imm	
	Branch <	SB	BLT rs1,rs2,imm	
	Branch ≥	SB	BGE rs1,rs2,imm	
	Branch < Unsigned	SB	BLTU rs1,rs2,imm	
	Branch ≥ Unsigned	SB	BGEU rs1,rs2,imm	
Jump & Link	J&L	UJ	JAL rd,imm	
	Jump & Link Register	UJ	JALR rd,rs1,imm	
Synch	Synch thread	I	FENCE	
	Synch Instr & Data	I	FENCE.I	
System	System CALL	I	SCALL	
	System BREAK	I	SBREAK	
Counters	ReaD CYCLE	I	RDCYCLE rd	
	ReaD CYCLE upper Half	I	RDCYCLEH rd	
	ReaD TIME	I	RDTIME rd	
	ReaD TIME upper Half	I	RDTIMEH rd	
	ReaD INSTR RETired	I	RDINSTRET rd	
	ReaD INSTR upper Half	I	RDINSTRETH rd	

Figure 1.3: Supported instructions under RISC-V 'I' Extension

Load and store instructions, such as LW and SW, offer memory interactions between the core and the memory. Control flow is managed through branch (BEQ, BNE, etc.) and jump instructions. One of the features of the 'I' extension is the arithmetic immediates. This feature is optimized for fast computations without extensive memory access, using the I-type format. The intrinsic structure of these instructions, with two register operands and a 12-bit immediate value, promotes efficient and rapid processing.

1.3.2 RV32M Extension

The 'M' extension of RISC-V deals with integer multiplication and division operations that are resource-intensive instructions that require dedicated hardware units. Instructions like MUL perform basic multiplication, while MULH, MULHSU, and MULHU cater to more subtle multiplication needs, producing the upper half of the multiplication result, and serving both signed and unsigned operations. Figure 1.4 [6] shows the instruction set of the 'M' extension and Table 1.2 [6] depicts all the supported instructions under this extension. This granularity of the 'M' extension ensures precise control over multiplication tasks, optimizing for performance and accuracy.

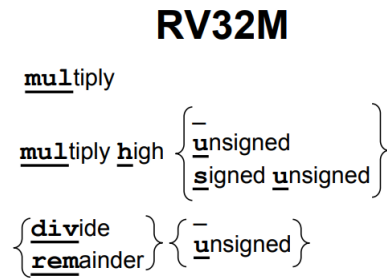


Figure 1.4: RV32M instruction set

With respect to division, the 'M' extension offers both signed (DIV) and unsigned (DIVU) divisions, ensuring broad applicability. The remainder of the division which is more used in algorithmic tasks is addressed through REM (signed) and REMU (unsigned) instructions.

Optional Multiply-Divide Instruction Extension: RVM					
Category	Name	Fmt	RV32M (Multiply-Divide)		+RV{64,128}
Multiply	MULTiply	R	MUL	rd,rs1,rs2	MUL{W D} rd,rs1,rs2
	MULTiply upper Half	R	MULH	rd,rs1,rs2	
	MULTiply Half Sign/Uns	R	MULHSU	rd,rs1,rs2	
	MULTiply upper Half Uns	R	MULHU	rd,rs1,rs2	
Divide	DIVide	R	DIV	rd,rs1,rs2	DIV{W D} rd,rs1,rs2
	DIVide Unsigned	R	DIVU	rd,rs1,rs2	
Remainder	REMAinder	R	REM	rd,rs1,rs2	REM{W D} rd,rs1,rs2
	REMAinder Unsigned	R	REMU	rd,rs1,rs2	REMU{W D} rd,rs1,rs2

Table 1.2: Supported instructions under RISC-V 'M' Extension

1.3.3 RV32C Extension

The developed Dobby SoC also supports the RV32C instruction set. To take advantage of the regularity and robust programmability of the RV32 ISA, and to reduce code size in memory-constrained environments, the RV32C extension introduces a set of 16-bit compressed instructions.

To reduce the code size, older ISAs increased the number of supported instructions and instruction formats. For example, adding shorter instructions with two operands instead of three, smaller immediate fields, and so on. ARM and MIPS introduced entirely new instruction sets like microMIPS, ARM Thumb, and Thumb-2 plus MIPS16 to achieve this. However, these

approaches created a lot of challenges for the processor, the compiler, and assembly language programmers.

RISC-V adopts a new approach to address the need for reduced code size. By mapping every short 16-bit instruction to a single standard 32-bit instruction, the existing RV32 ISA can be retained, while allowing for reduced memory usage and denser codes. Figure 1.5 [6] presents a graphical representation of the RV32C extension instruction set. Compressed floating point instructions are not under the scope of this project work.

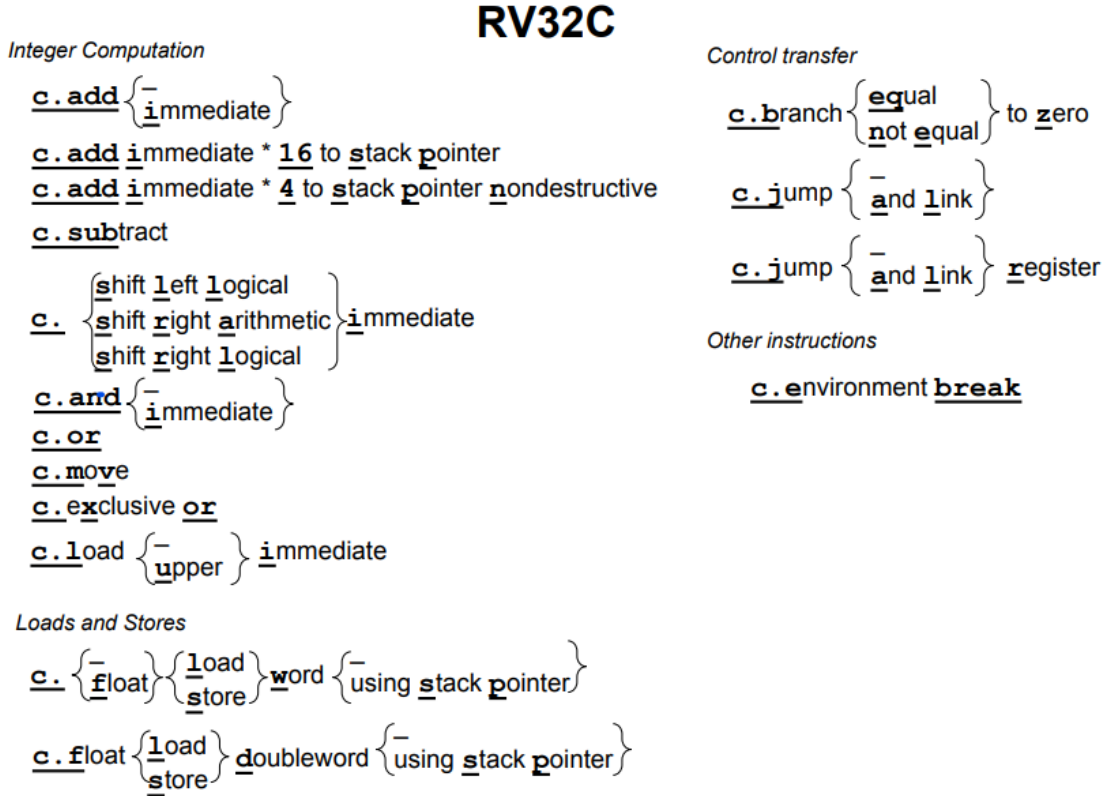


Figure 1.5: RV32C instruction set

The opcode space of typical 32-bit RISC-V instructions is divided into smaller opcode spaces for compressed instructions. This allows instructions to be represented using fewer bits, saving memory space. Only a subset of registers in the local register file within the core can be accessed using compressed instructions. Furthermore, immediate operands tend to be smaller in this case. The PRAM incorporated within the Dobby SoC can hold 32-bit instructions. If compressed instructions were to be used, two 16-bit instructions could be stored in the same space, promoting better memory usage.

It is important to note that not all 32-bit instructions are available in compressed format due to the constraints of the encoding scheme. Complex and very rarely used 32-bit instructions are typically not included in the RV32C instruction set. Table 1.3 [6] presents a list of compressed instructions supported by the Dobby SoC core along with their equivalent RISC-V instructions.

Category	Name	Fmt	RVC		RISC-V equivalent	
Loads	Load Word	CL	C.LW	rd',rs1',imm	LW	rd',rs1',imm*4
	Load Word SP	CI	C.LWSP	rd,imm	LW	rd,sp,imm*4
Stores	Store Word	CS	C.SW	rs1',rs2',imm	SW	rs1',rs2',imm*4
	Store Word SP	CSS	C.SWSP	rs2,imm	SW	rs2,sp,imm*4
Arithmetic	ADD	CR	C.ADD	rd,rs1	ADD	rd,rd,rs1
	ADD Immediate	CI	C.ADDI	rd,imm	ADDI	rd,rd,imm
	ADD SP Imm * 16	CI	C.ADDI16SP	x0,imm	ADDI	sp,sp,imm*16
	ADD SP Imm * 4	CIW	C.ADDI4SPN	rd',imm	ADDI	rd',sp,imm*4
	SUB	CR	C.SUB	rd,rs1	SUB	rd,rd,rs1
	AND	CR	C.AND	rd,rs1	AND	rd,rd,rs1
	AND Immediate	CI	C.ANDI	rd,imm	ANDI	rd,rd,imm
	OR	CR	C.OR	rd,rs1	OR	rd,rd,rs1
	eXclusive OR	CR	C.XOR	rd,rs1	AND	rd,rd,rs1
	MoVe	CR	C.MV	rd,rs1	ADD	rd,rs1,x0
	Load Immediate	CI	C.LI	rd,imm	ADDI	rd,x0,imm
	Load Upper Imm	CI	C.LUI	rd,imm	LUI	rd,imm
Shifts	Shift Left Imm	CI	C.SLLI	rd,imm	SLLI	rd,rd,imm
	Shift Right Ari. Imm.	CI	C.SRAI	rd,imm	SRAI	rd,rd,imm
	Shift Right Log. Imm.	CI	C.SRLI	rd,imm	SRLI	rd,rd,imm
Branches	Branch=0	CB	C.BEQZ	rs1',imm	BEQ	rs1',x0,imm
	Branch≠0	CB	C.BNEZ	rs1',imm	BNE	rs1',x0,imm
Jump	Jump	CJ	C.J	imm	JAL	x0,imm
	Jump Register	CR	C.JR	rd,rs1	JALR	x0,rs1,0
Jump & Link	J&L	CJ	C.JAL	imm	JAL	ra,imm
	Jump & Link Register	CR	C.JALR	rs1	JALR	ra,rs1,0

Table 1.3: Supported compressed instructions and their equivalent RV32I instruction

2 Design and Implementation

This chapter describes the design and implementation aspects of modules constituting the Dobby SoC. Efforts made towards achieving the design goals and complications resolved during the design phase are presented.

The Dobby SoC comprises two major components: the RISC-V core and the “top_memory_controller” unit as depicted in [Figure 2.1](#) below. The “top_memory_controller” unit handles all memory operations to and from the core.

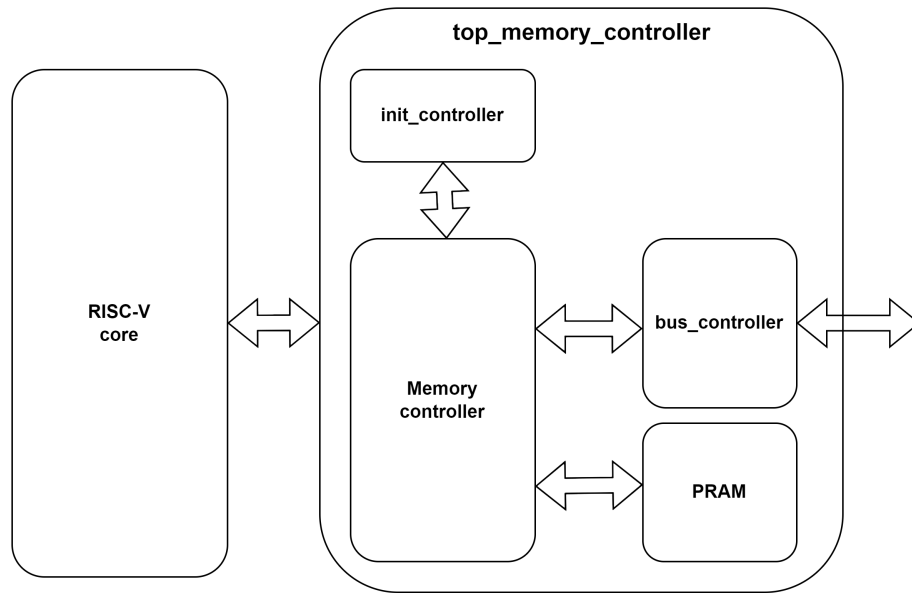


Figure 2.1: Block diagram of Dobby SoC

2.1 RISC-V Core

The RISC-V Dobby core implements the RV32IMC ISA, featuring an optimized design to execute integer, multiplication, compression, and control instructions. The core contains multiple units including a PC unit, fetch unit, decode unit, execute unit ([Arithmetic Logic Unit \(ALU\)](#)), [Register File \(RF\)](#), [Control and Status Register \(CSR\)](#) unit, interrupt controller, and control unit. During operation, the core fetches instructions from memory based on the

current PC. It then decodes instructions and executes them via the ALU, updating the PC to fetch the next instruction. The core also generates control signals for the memory controller to read from or write to PRAM and external memory. The riscv_top module connects the Dobby core units together based on their signal interfaces for seamless functioning. Below is a closer look at each functional unit and its cohesive functions:

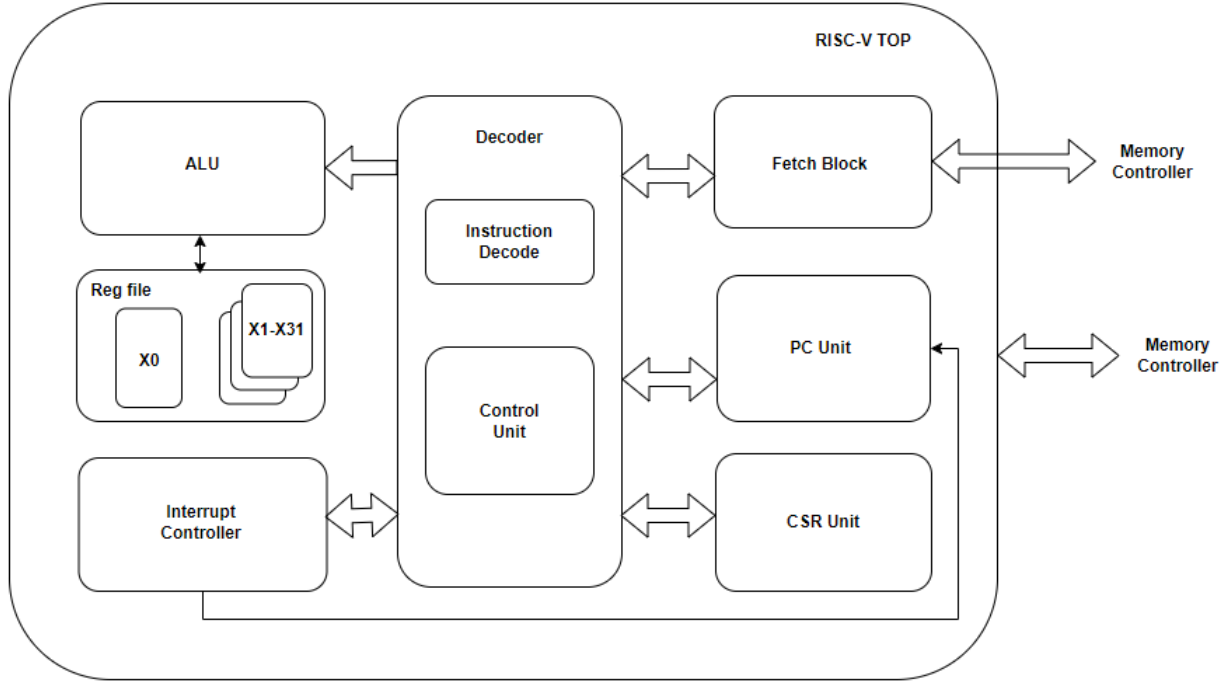


Figure 2.2: Block diagram of Dobby Core

2.1.1 Fetch Unit

The fetch unit integrated within the "riscv_top" module, plays a pivotal role in retrieving instructions and data from the PRAM and external memory. Using the address from the PC unit, the Fetch unit communicates with the memory controller to fetch the required instruction. It then passes this instruction to the decode unit for interpretation. It produces vital control signals for the memory controller, encompassing enable signals, target address, and PC address.

2.1.2 PC Unit

The PC unit module specifies the progress of the program counter, the main determinant of the execution flow. In particular, its multiple input signals and various decoders and control flags drive the module, providing the address of the next instruction. The essence of this module is its ability to automatically increment the program counter. During normal operation, the PC is incremented by 4 bytes (or 32 bits) and moves to the next instruction in memory. Dobby core also supports compressed instructions, 16-bit (or 2-byte) instructions. When the

PC unit detects such a compressed instruction, the PC simply makes it 2 bytes, ensuring that the next instruction is correctly received.

For branch-jump instructions, the PC unit determines the next value of the PC using the current value of the registers and the conditions specified by the control signals. On initialization or in special cases (such as a system reset), the PC unit sets the program counter to a default memory location, commonly referred to as the reset vector. In the code provided, this reset location is defined as `PC_RESET_VALUE = 32'h0008`.

In the event of interrupts or exception, the PC will effectively terminate the current process and directs it to the Interrupt Service Routine or exception handler. Acknowledgement of the interrupt is also provided by this unit to the external interface when the PC is set to the exception handler address. The block diagram of the PC unit is shown in [Figure 2.3](#) and its coverage report is depicted in [Figure 2.4](#).

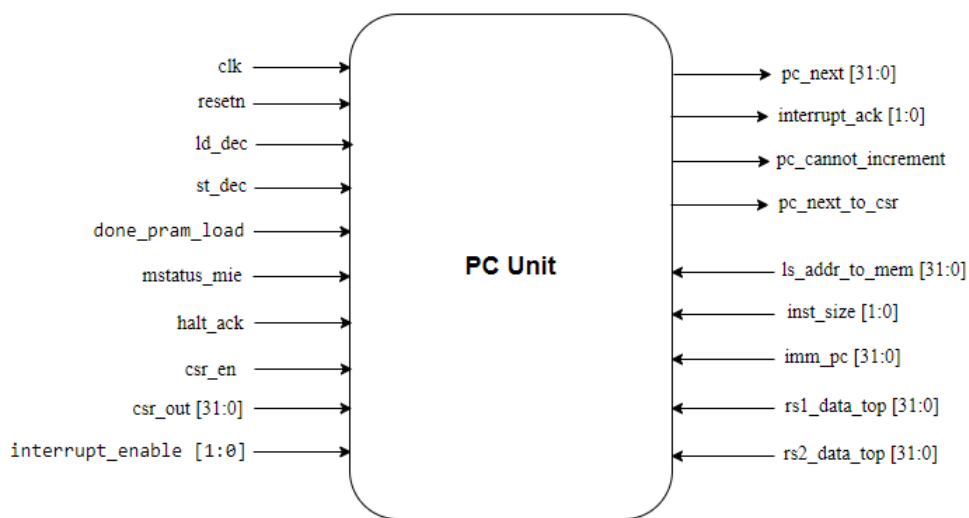


Figure 2.3: Block diagram of the PC Unit


Module	Block
tb_pc_unit	100% 26 / 26
Line	e:/home2/vlsi00/prz-root/icpro_teamspace/prz22/workspaces/group13/koke22/units/pc_unit/source/tb/ve
1	// Company : tud
2	// Author : koke22
3	// E-Mail : <email>
4	//
5	// Filename : tb_pc_unit.v
6	// Project Name : prz
7	// Subproject Name : qf28_template
8	// Description : <short description>
9	//
10	// Create Date : Sun Sep 10 15:50:27 2023
11	// Last Change : \$Date: 2023-09-12 18:05:32 +0200 (Tue, 12 Sep 2023) \$
12	// by : \$Author: koke22 \$
13	//-----
14	timescale 1ns/10ps
15	module tb_pc_unit (
Coverage Report: Uncovered Blocks 	
This module has 100% block coverage.	
Marking	

Figure 2.4: Coverage report for the PC unit

2.1.3 Decoder

The RISC-V RV32IMC decoder handles instructions from the integer (I), multiply/divide (M), and compressed (C) extensions of the RISC-V ISA. It takes either a 32-bit base instruction or a 16-bit compressed instruction as input and determines the instruction type based on the opcode. For standard 'I' instructions, it decodes the register operands, immediates, and generates controls like ALU sub-opcodes, load/store sizes, functional unit selection, and others to feed the control units. For compressed C instructions, it utilizes a compression decoder to expand the 16-bit instruction to its 32-bit equivalent before decoding. For multiply/divide M instructions, it activates the multiplier/divider functional unit. Additional functionality of the decoder includes calculating PC offsets for branches and jump instructions, handling system instructions like ECALL and MRET, and outputting instruction size and control flags.

The decoder is a combinational circuit. The key implementation aspects include the decoding logic, operand extraction, immediate calculation, compression expansion, operand mux-ing based on decoded controls, sub-opcode generation, and control signal production. In summary, the RV32IMC decoder efficiently decodes RISC-V instructions from the base I, compressed C, and multiply M extensions into simplified controls and operands to feed the later units in the Dobby core. The block diagram of the decoder unit is shown in [Figure 2.5](#) and its coverage report is depicted in [Figure 2.6](#).

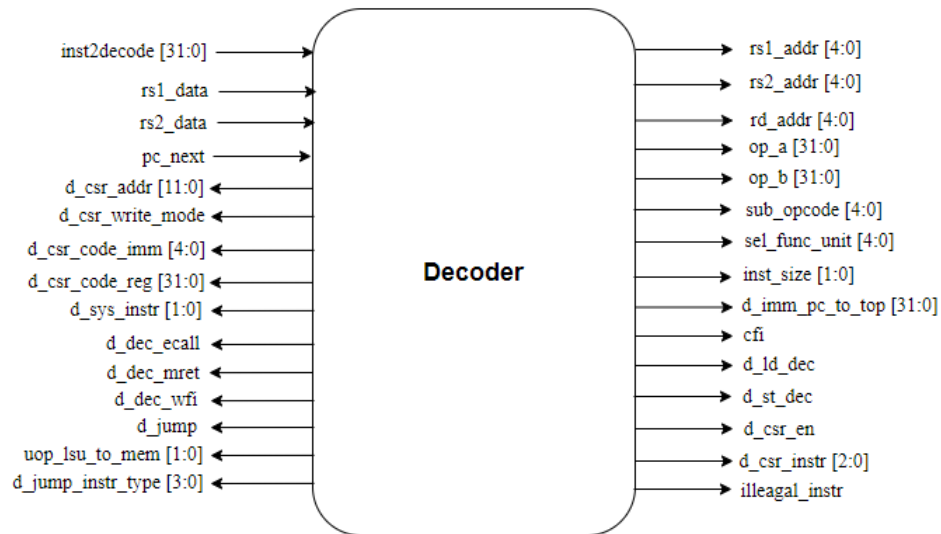


Figure 2.5: Block diagram of the Decoder unit

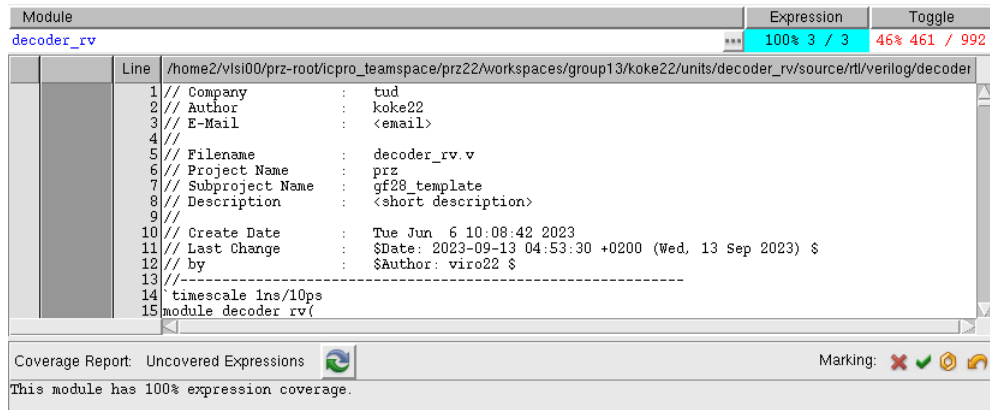


Figure 2.6: Coverage report for the Decoder unit

2.1.4 Compression Decoder

When the decoder encounters a compressed instruction, it enables the compression decoder unit. The compression decoder transforms the 16-bit compressed instruction into its corresponding 32-bit version, as depicted in Table 1.3. Subsequently, the decoder decodes this expanded instruction to generate the necessary control signals. Figure 2.7 presents the coverage report for the compression decoder unit.

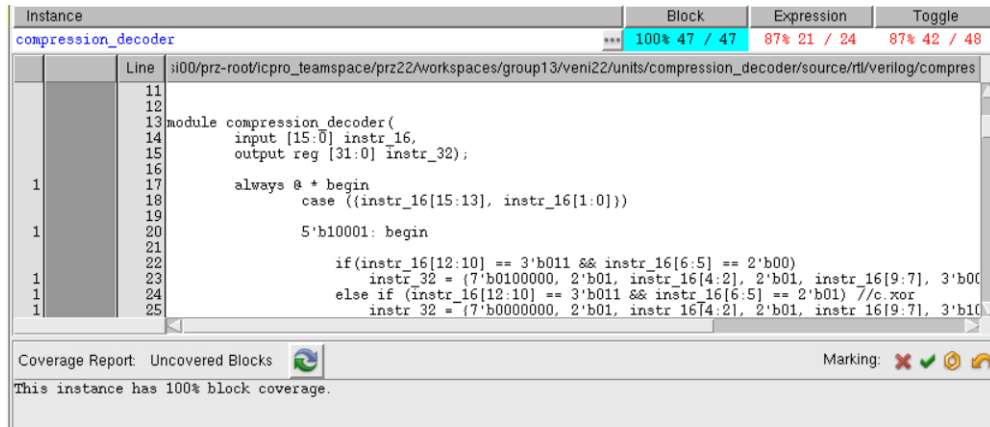


Figure 2.7: Coverage report for the Compression decoder unit

2.1.5 ALU

The ALU can perform a wide range of arithmetic and logical operations specified by the RISC-V instruction set for the Dobby core. This ALU operates on two inputs, 'alu_lhs' and 'alu_rhs' and uses a 5-bit opcode to determine the specific operation to be carried out. For arithmetic operations like addition (ADD) and subtraction (SUB), it directly performs the calculations. It also handles operations such as AND, OR, XOR, and various shift operations. However its capabilities extend beyond arithmetic and logic; it also handles more complex multiplication operations like MUL, MULH, MULHSU and MULHU. Additionally, it effectively manages division (DIV and DIVU) as well as modulus operations (REM and REMU) taking into

account special cases like division by zero or overflow scenarios. ALU unit also handles both signed and unsigned operations efficiently.

The block diagram of the ALU unit is shown in Figure 2.8 and its coverage report is depicted in Figure 2.9.

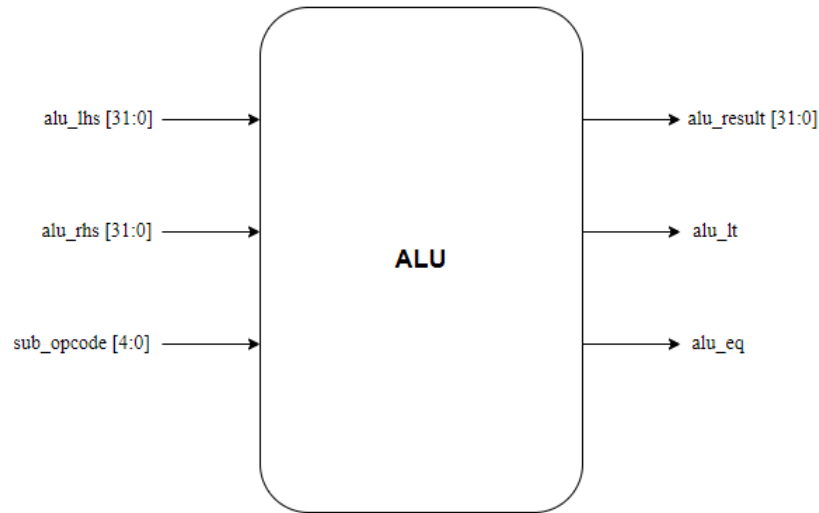


Figure 2.8: Block diagram of the ALU

Module	Block	Expression
alu	100% 35 / 35	100% 6 / 6

Line	File: /home2/vlsi00/prz-root/icpro_teamspace/prz22/workspaces/group13/koke22/units/alu/source/rtl/verilog/alu.v
1	// Company : tud
2	// Author : koke22
3	// E-Mail : <email>
4	//
5	// Filename : alu.v
6	// Project Name : prz
7	// Subproject Name : gf28_template
8	// Description : <short description>
9	//
10	// Create Date : Sat May 13 23:03:03 2023
11	// Last Change : \$Date: 2023-09-09 17:06:24 +0200 (Sat, 09 Sep 2023) \$
12	// by : \$Author: viro22 \$
13	//-----
14	timescale 1ns/10ps
15	module alu (

Coverage Report: Uncovered Blocks	Marking: X ✓ ⚠ ↺
-----------------------------------	------------------

This module has 100% block coverage.

Figure 2.9: Coverage report for the ALU unit

2.1.6 Register file

The 'riscv_regfile' module is an integral part of the RISC-V microarchitecture, which acts as a register file for the CPU. This module contains 32 registers, each 32-bit wide, labelled rf0 through rf31. The x0 register in the reg file is hardwired to always return zero. It simplifies some computer programs and reduces the need for immediate zero constants in the instruction set. By synchronizing the clock with an active low reset, the module ensures that the register

values are zero at reset. The write enable signal determines whether writes to registers are allowed. The module provides the capability to read from two registers (ra0_i and rb0_i) and write to one (rd0_i) in one clock cycle, which includes read-write capabilities simultaneously. Corresponding read values are output displayed as ra0_value_o and rb0_value_o, when the value to be written is given by rd0_value_i.

The block diagram of the reg file unit is shown in Figure 2.10 and its coverage report is depicted in Figure 2.11.

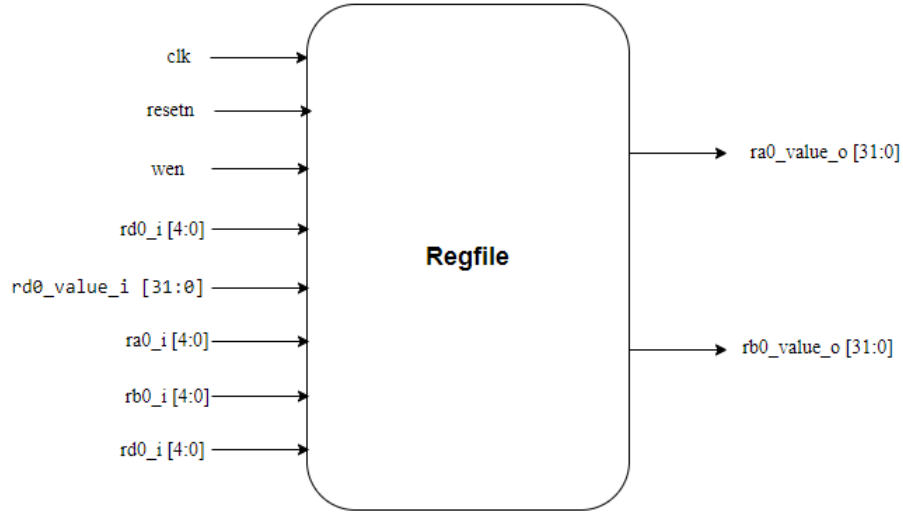


Figure 2.10: Block diagram of the Regfile

Module	Block	Expression
riscv_regfile	97% 36 / 37	100% 3 / 3
Line	home2\visi00\prz-root\icpro_teamspace\prz22\workspaces\group13\koke22\units\riscv_regfile\source\rtl\	
1	/*	
2	// Company : tud	
3	// Author : koke22	
4	// E-Mail : <email>	
5	//	
6	// Filename : riscv_regfile.v	
7	// Project Name : prz	
8	// Subproject Name : gf28_template	
9	// Description : <short description>	
10	//	
11	// Create Date : Tue Jun 6 17:29:15 2023	
12	// Last Change : \$Date: 2023-09-10 15:06:32 +0200 (Sun, 10 Sep 2023) \$	
13	// by : \$Author: viro22 \$	
14	//-----	
15	timescale 1ns/10ps	
Coverage Report: Uncovered Blocks		
This module has 100% block coverage.		

Figure 2.11: Coverage report for the reg file unit

2.1.7 Control Status Registers

This RISC-V processor implementation consists of 5 control status registers, MISA, MTVEC, MSTATUS, MCAUSE and MEPC. These are used in handling machine-mode interrupts and implement machine-level privileged instructions. The functionality and implementation of these registers is given below:

- MISA - Machine Instruction Set Architecture: Provides information on the architecture of the design. Here, it is coded to represent RV32IMC. This register is read-only.
- MTVEC - Machine Trap-Vector: Specifies the base address to reset the PC to when a trap is caused by an interrupt into machine mode. Here, it is coded to the value 0. This register is read-only.
- MSTATUS - Machine Status Register: Describes the status of the machine during interrupt enables. Since interrupts cannot be enabled independently, the MIE bit of the mstatus register decides whether an incoming interrupt request can be acknowledged. If the request is accepted, the MIE bit is reset and further interrupts cannot be accepted until the machine returns from the interrupt handler. This register is read- and write-enabled.
- MEPC - Machine Exception Program Counter: Holds the value of PC when a trap occurs. Whenever an interrupt request is accepted, or an environment transfer occurs, the mepc register holds the next value of PC, that the machine must jump to, once the machine returns from the sequence. This register is read- and write-enabled.
- MCAUSE - Machine Cause: Holds information about the machine when a trap occurs. Here, a bit is set whenever an environment call or an interrupt occurs. This register is read- and write-enabled.

The privileged instructions that the CSRs support are given below:

- CSRRW and CSRRWI: Atomically read the specified CSR and write to it based on an immediate or a register value.
- CSRRC and CSRRCI: Atomically read the specified CSR and clear particular bits of the CSR using a mask specified in an immediate or a register value.
- CSRRS and CSRRSI: Atomically read the specified CSR and set particular bits of the CSR using a mask specified in an immediate or a register value.
- MRET: Machine mode return. This is used to return from a machine mode trap back to the normal execution using the PC value stored in MEPC. Using this instruction sets the MIE bit of the MSTATUS register again.
- WFI: Wait for Interrupt. This is used to halt all execution till an interrupt occurs. Once the interrupt is acknowledged, the machine moves to the interrupt handler and upon return continues normal execution.

2.2 Top Memory Controller

This unit encapsulates the memory-related units including the initialization controller, memory controller, in-built program memory, and bus controller. Creating a dedicated unit to manage memory requests helps in achieving a higher level of abstraction within the SoC. It streamlines the verification process for the core, memory components, and in turn the SoC.

2.2.1 Memory Controller

The memory controller unit connects the core with the internal program memory and external interface through the bus controller. It handles the fetching of instructions as well as the loading/storing of data to/from the Dobby core's register file. Depending on the specific access type (instruction fetch or load-store operation), the memory controller generates the required control signals based on the input address. These signals are then transmitted to both the bus controller and the internal PRAM. It is composed of two units namely, the trim unit and the load-store controller. The block diagram of the memory controller unit is depicted in Figure 2.12. Figure 2.13 presents the coverage report for the memory controller unit.

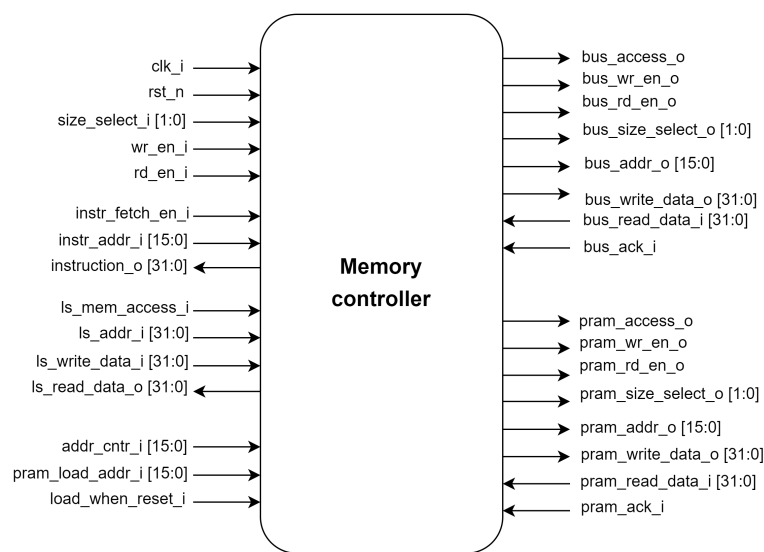


Figure 2.12: Block diagram of the Memory Controller

Module	Types	Self Total	Cumulative Total
memory_controller	bet	98% (49 / 50)	99% (80 / 81)
load_store_controller	bet	100% (21 / 21)	100% (21 / 21)
trim_unit	bet	100% (10 / 10)	100% (10 / 10)

Figure 2.13: Coverage report for the memory controller unit

2.2.2 Program Memory

The Dobby SoC features a 16KB program memory, which is constructed using four SRAM macros of the type "HM_1P_GF28SLP_1024x32_1cr," with each macro capable of storing 1K x

32 bits. During the initialization phase, the PRAM is loaded with initial configurations, a task facilitated by both the memory controller and the initialization controller. The PRAM supports both read and write operations, completing them in a single cycle. It accommodates the writing of bytes, half-words, and words into the macros based on the input address and "size_select" signals.

In this implementation, each memory location corresponds to one byte location of all the 4 SRAM macros. This way, each 32 bit word is stored across all the 4 SRAM macros. From the 16 bit address that the PRAM receives, only bits [13:0] are used to address the 64 Kb of memory.

Table 2.1: PRAM Address Mapping to SRAM Macros

Address	SRAM 0		SRAM 1		SRAM 2		SRAM 3	
[3:0]	Address	Byte	Address	Byte	Address	Byte	Address	Byte
0000	[13:4]	[7:0]	[13:4]	[7:0]	[13:4]	[7:0]	[13:4]	[7:0]
0001	[13:4]	[15:8]	[13:4]	[7:0]	[13:4]	[7:0]	[13:4]	[7:0]
0010	[13:4]	[15:8]	[13:4]	[15:8]	[13:4]	[7:0]	[13:4]	[7:0]
...								
1101	[13:4]+1	[7:0]	[13:4]	[31:24]	[13:4]	[31:24]	[13:4]	[31:24]
1110	[13:4]+1	[7:0]	[13:4]+1	[7:0]	[13:4]	[31:24]	[13:4]	[31:24]
1111	[13:4]+1	[7:0]	[13:4]+1	[7:0]	[13:4]+1	[7:0]	[13:4]	[31:24]
...								

The mapping of the SRAM macros to the input address is shown in [Table 2.1](#), while [Figure 2.14](#) illustrates the block diagram of the designed PRAM. [Figure 2.15](#) presents the coverage report for the program memory.

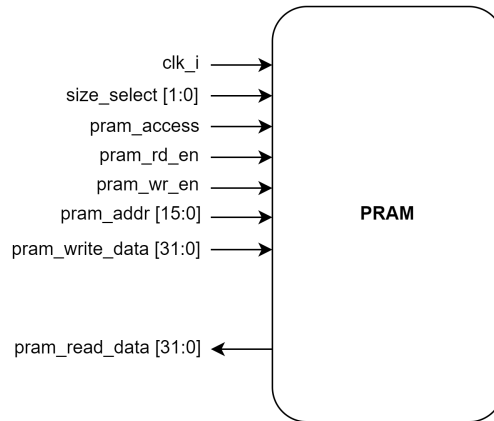


Figure 2.14: Block diagram of the Program Memory

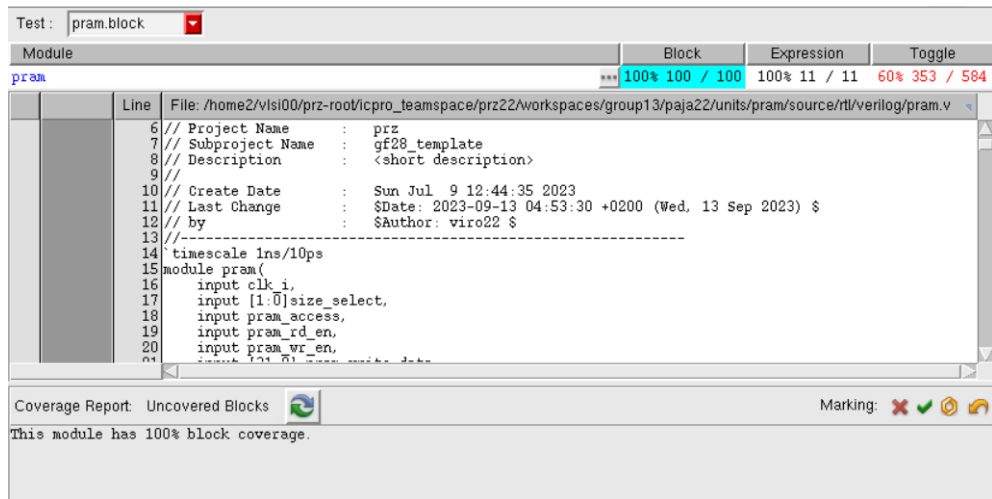


Figure 2.15: Coverage report for the PRAM unit

2.2.3 Bus Controller

The bus controller unit handles all memory operations to/from the core and external interface. A bus transfer typically involves a request phase and a data phase. The slave can insert any number of wait cycles. Based on the availability of slave(s), the Dobby SoC can maintain its signals at the external interface. PC updation is halted until the corresponding bus transfer is completed. The master and slave signals are registered to improve timing performance. Sub-word accesses (1 or 2 bytes) are implemented via the “O_BUS_SIZE” port. A memory transfer request within the address range of 0x4000 to 0x1FFFF, or a forced external access, enables the bus controller unit. Figure 2.16 depicts the block diagram of the bus controller unit. Waveforms in Figure A.1 and Figure A.2 depict the capability of the bus controller to handle wait cycles induced by the slave during bus write and read operations. Figure 2.17 presents the coverage report for the bus controller unit.

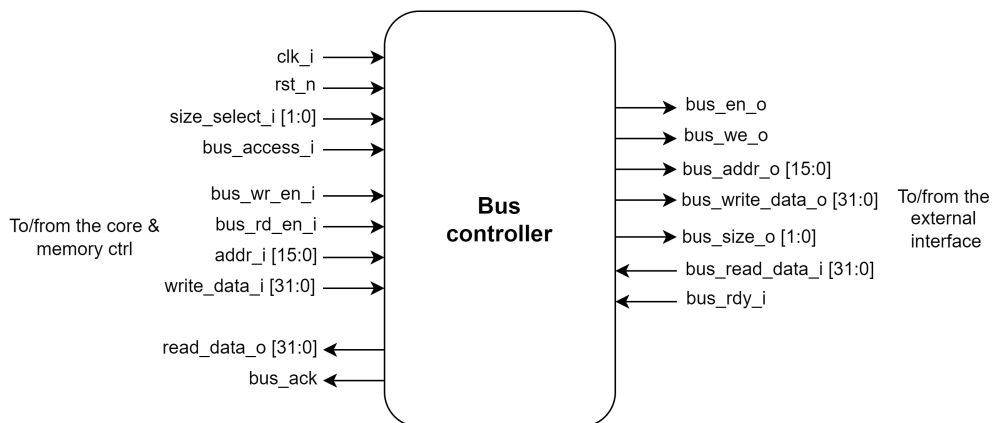


Figure 2.16: Block diagram of the bus controller unit

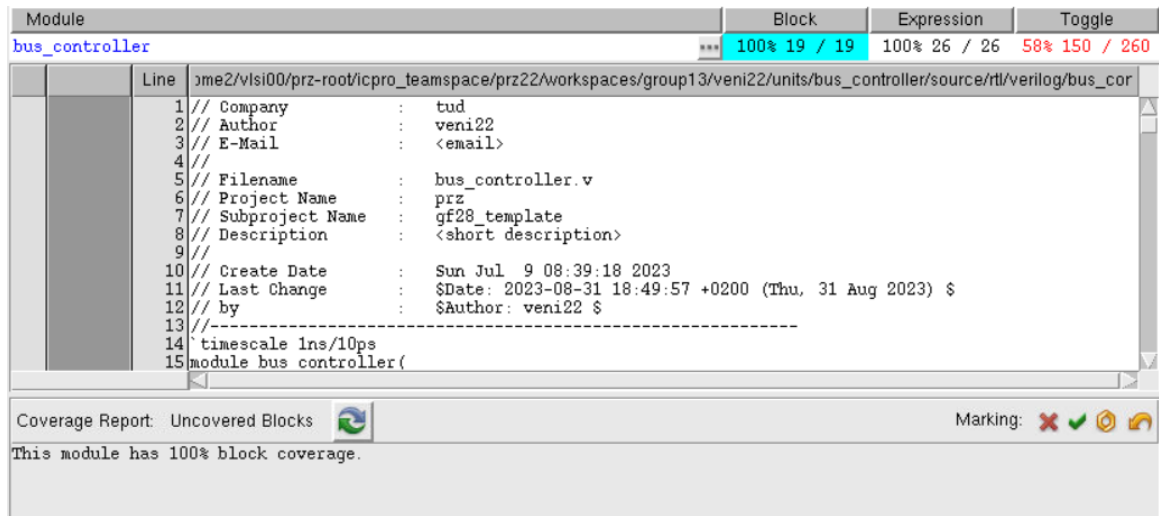


Figure 2.17: Coverage report for the bus controller unit

2.2.4 Initialization Controller

As per the specifications provided, the program memory must be loaded in the event of a hard reset. Whenever the reset pin is de-asserted, memory content from addresses 0x0000 to 0x3FFF is read from the external interface and written to the corresponding memory locations within the PRAM. The init_controller module incorporates an [Finite State Machine \(FSM\)](#) which helps in realizing this functionality. [Figure 2.18](#) depicts the FSM within the init_controller unit. Upon a reset, the FSM enters the LOAD state. In this state, the init_controller in conjunction with the memory_controller generates the necessary addresses, control signals, and read/write data signals that are transmitted to both the PRAM and bus_controller units.

The init_controller unit also considers any possible wait cycles induced by the slave during program memory initialization. In the absence of wait cycles (the slave is always ready), it takes four clock cycles to write a single word from the external interface into the PRAM. Furthermore, the Dobby core is disabled during this state. As soon as all the memory locations within the PRAM are loaded or when IRQ0 is activated, the FSM enters the IDLE state. This indicates the completion of memory initialization. In the absence of IRQ0 events, memory initialization consumes 16,384 clock cycles. The "load_when_reset" signal is cleared which in turn indicates the Dobby core to perform its intended functions. [Figure 2.19](#) presents the coverage report for the init_controller unit.

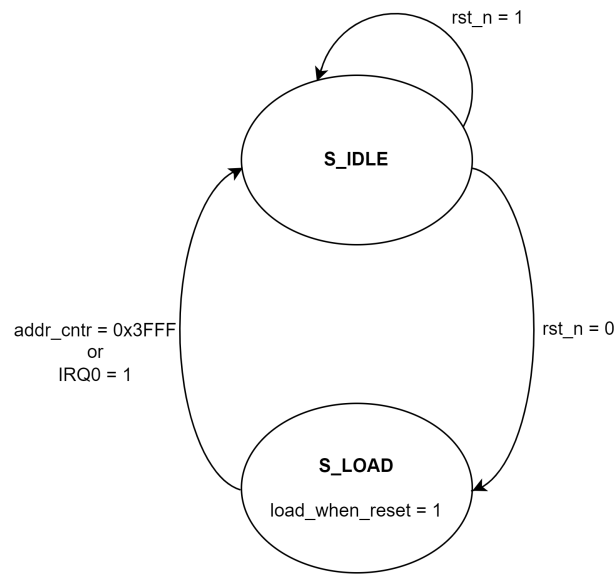


Figure 2.18: FSM in the initialization controller

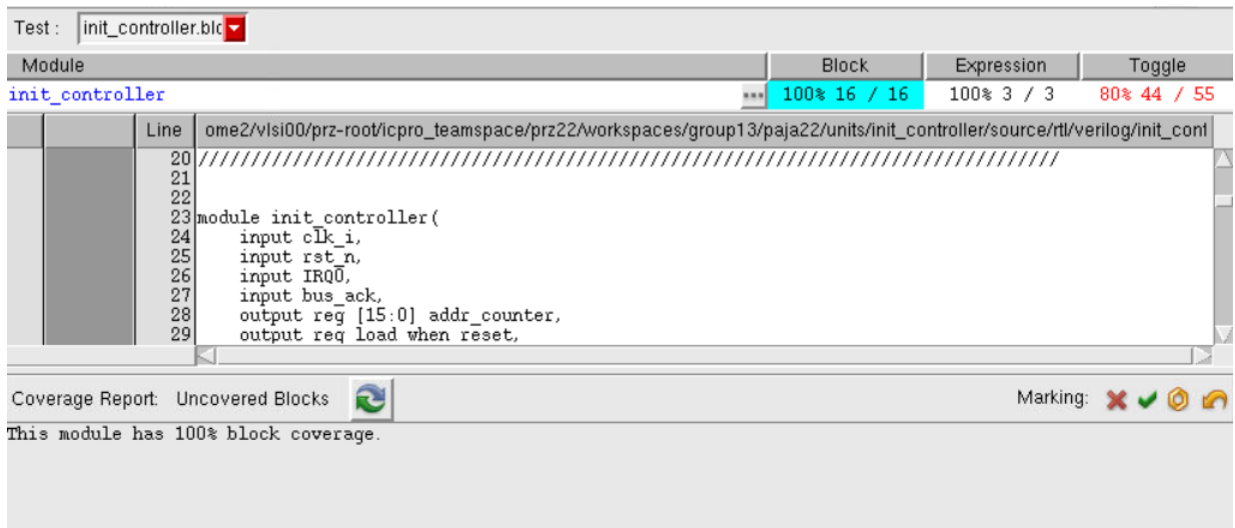


Figure 2.19: Coverage report for the initialization controller

2.3 Data and control paths of the SoC

Figure 2.20 represents the data and control paths in the top level of the Dobby SoC. Here, all lines represented in black are data paths and all in blue are control paths. The register stages are represented by a dotted line running through the registers in each path. This design has two stages, one for the signals to the memory control and the other for the data to the core.

It should be noted that not all paths are active at once. Different instructions enable different paths for the correct processing inside the core.

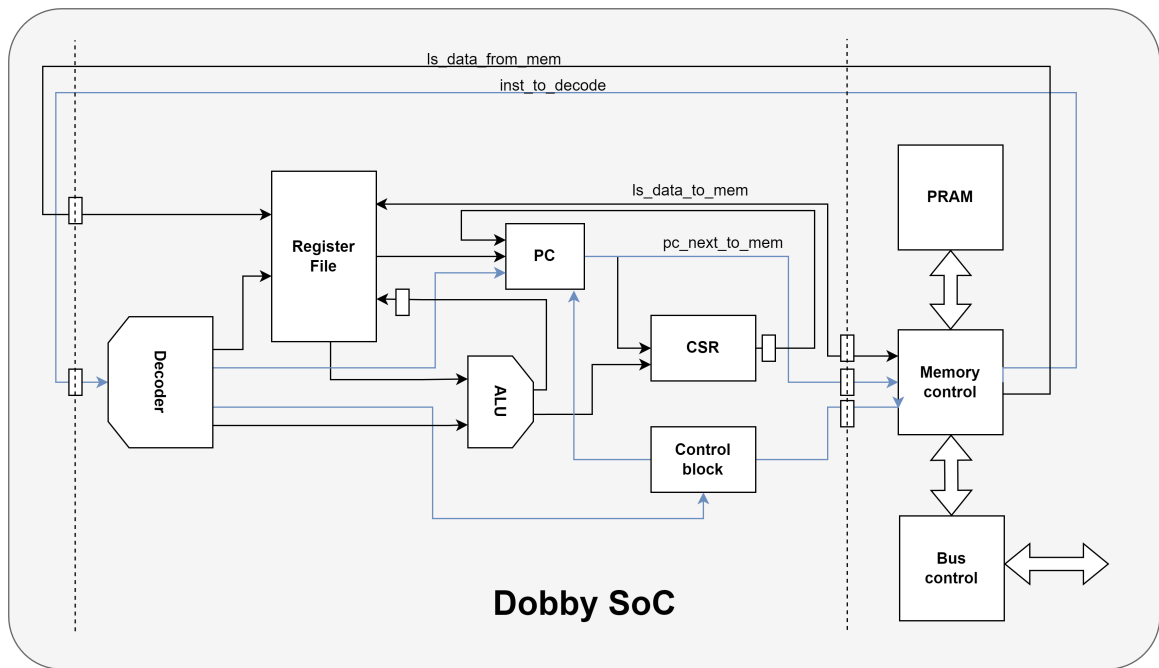


Figure 2.20: Datapath of the Core

2.4 Design optimizations to meet the design goals

2.4.1 Single Cycle Design

The original conception of the core design was aimed to be a **Single-cycle processor**, with an **optimization on speed** (throughput). This would be capable of executing all RV32IMC instructions in a single clock cycle with the exception of external fetches and load-store operations. This logic was implemented with the main goal of achieving reliable functionality while avoiding any possible combinational loops.

However, the trade-off of this design was an insubstantial operating frequency of around **60 MHz** (clock period of over **15 ns**).

2.4.2 Two-stage Design

To overcome this issue and achieve a higher frequency of operation, the implementation was modified to be a two-stage design, with all control signals being registered between the RISC-V core and top_memory_controller unit.

Pipelining this design was fairly simple, considering the fact that the SRAM macros have a delay of one clock cycle.

The resulting frequency of operation from this optimization was around **90 MHz** (clock period of **11 ns**).

2.4.3 Breaking down the critical path

The next stages of optimization were done by carefully examining the critical timing path in the design. The current design has two stages of registers; one for the control signals between the core and the memory controller, and the other for the data at the output of the PRAM.

However, the critical path showed that the control signals formed a complete loop, from the output of the registers to the memory controller, back to the input of the same registers. This path was then broken down at the output of the PRAM, as controlling the PRAM output only requires the information already sampled in the previous clock cycle. This optimization led to a frequency of **125 MHz** (clock period of **8 ns**).

2.4.4 Optimizing the critical path

At this point, the critical path could not be broken down any further without the addition of more register stages. The idea behind further optimizations is that any logic within the critical path that can be calculated independent of the data on the critical path, should instead be performed in parallel.

This reduces the delay of additional gate logic, which can add up to a significant amount considering larger paths in designs with only a few stages.

Iteratively optimizing the design in this manner led to a maximum operating frequency of around **150 MHz** (clock period of **6.5 ns**).

It is important to note that the stated frequency figures are obtained from the logic synthesis reports.

2.4.5 Memory optimization

Considering compressed instructions, a challenge arises when there is an instruction the lower 16 bits of a 32-bit word, with the upper half of that word holding half of a subsequent normal instruction. This needs two PRAM reads for the execution of a single instruction. Solving this issue by retaining the previous half-word to reduce memory accesses is ineffective, when the instruction involves a jump/ change in PC.

Furthermore, there is an increase in the average number of clock cycles required per instruction when processing compressed instructions. In order to maintain the same level of throughput as normal instructions, the PRAM is converted to a byte-addressable format. This allows a 32-bit word to be retrieved from a half-memory location, thus optimizing memory access. However, there is higher power consumption as all the SRAM macros are simultaneously accessed.

2.4.6 Delay for each Instruction type

[Section 2.4.6](#) summarizes the total number of clock cycles that each instruction takes to execute.

Instruction	PC address location	Total Clock Cycles
Single cycle/Immediate	PRAM	2
Store to PRAM	PRAM	3
Load from PRAM	PRAM	4
Load from EXT	PRAM	6
Store to EXT	PRAM	6
Single cycle/Immediate	EXT	4
Store to PRAM	EXT	5
Load from PRAM	EXT	6
Load from EXT	EXT	8
Store to EXT	EXT	8

Table 2.2: Total clock cycles for all instruction types

3 Simulation and Verification

This chapter discusses the simulation of the Dobby SoC and the verification of its functionality. The verification of the individual blocks is first performed for functionality as well as coverage, as shown in the reports of the previous chapter. The Cadence NCSim tool [8] is used for the verification of this design.

3.1 Testbench

To verify the functionality of the core, a testbench environment is set up as shown in [Figure 3.1](#). Here, the Dobby SoC is connected with two external peripherals, an external memory and the DNN accelerator provided. The Verif_top interface handles the external bus logic and connects the respective peripheral with the core depending on the bus address coming from the core.

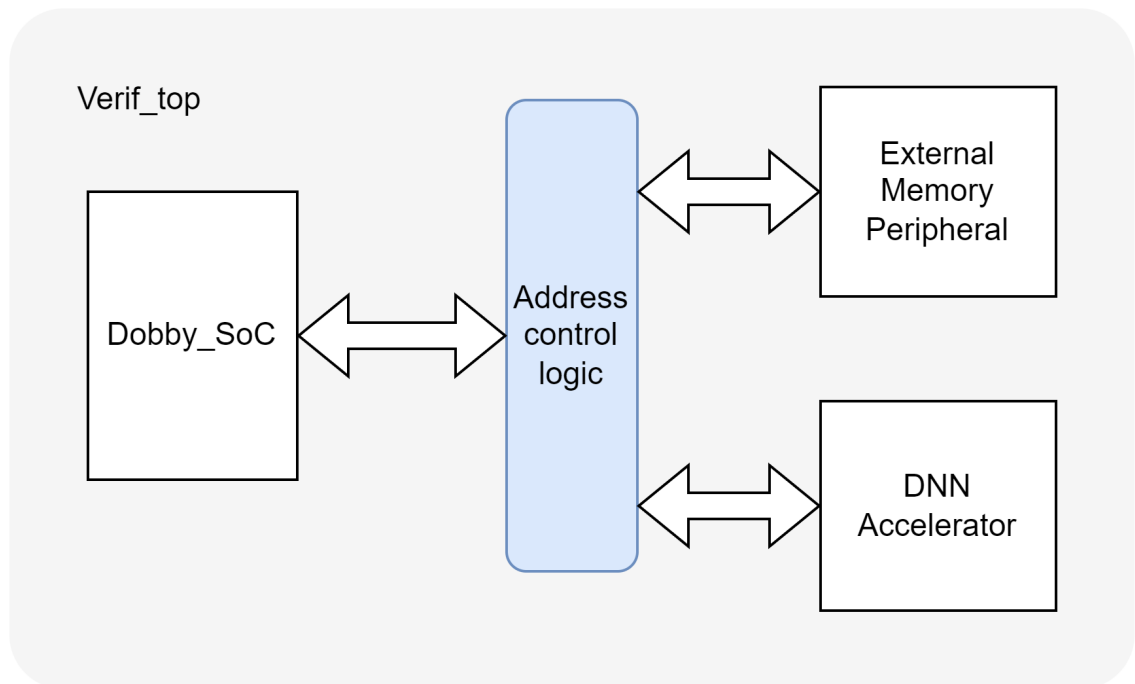


Figure 3.1: Overview of Testbench Environment

The peripheral external memory consists of a 16KiloByte RAM which is byte addressable. Upon receiving a request, the peripheral samples the control signals including the data word size and the address. The first RDY acknowledgement is passed back to the core once sampled. In the next cycle, the second RDY acknowledgement comes along with the read data from the RAM, or when the write data from the bus is stored in the memory.

Although the RAM is byte addressable since the data bus is 32 bits wide, 4 bytes from the given address are read from the RAM. Size select is used during the write operation to decide whether a word, half-word or byte is stored.

The second peripheral connected is the DNN accelerator. The current implementation only supports the address mapping to the DNN accelerator. During initialization, when the internal PRAM of the core is being loaded from the external memory, addresses 0x0 to 0x3fff correspond to the first 4 KB of the external memory's RAM. However, after loading is done, when external access is requested to this address range of 0x0 to 0x3fff, it is directed to the DNN accelerator. Addresses in the range 0x4000 to 0xffff are only sent to the external memory.

3.2 Simulation Setup

To simulate this design, the Unit "verif_top" is used. The path to access the test case file and memory file in the SVN repository is *units/verif_top/simulation/ncsim/rtl_tc0*.

Here, the program code is written to the *ext_mem1.mem* file. The peripheral memory reads this file and loads the program code into the PRAM during the initialization of the simulation.

The file *testcase.v* contains the stimuli for the simulation of the program. The first 16384 clock cycles are allocated for loading the PRAM, after which the load_done register is set to 1 in the testbench. The simulation then runs for another 1000 clock cycles as a default. This number can be changed if the expected time for the program execution is larger.

Currently, the *ext_mem1.mem* file contains program code for a simple matrix multiplication. This program code can be overwritten to test any new code/testcase.

3.3 Verification Test cases

3.3.1 RV32IM ISA with ASM code

To verify the complete functionality of the design, an ASM code consisting of all the instructions in the ISA is used. Below is the summary of the most important functionalities and instructions being tested.

PRAM load

Upon initialization, the PRAM is loaded with the entire program code from the external memory through the peripheral interface.

Load from Bus

Next, instructions to load words, half words and bytes from the external memory are tested. As can be seen in [Figure 3.4](#), The bus behavior is similar to that of PRAM load, but here the data is sent only to the regfile. The connection between PRAM and EXT is disabled after loading is complete.

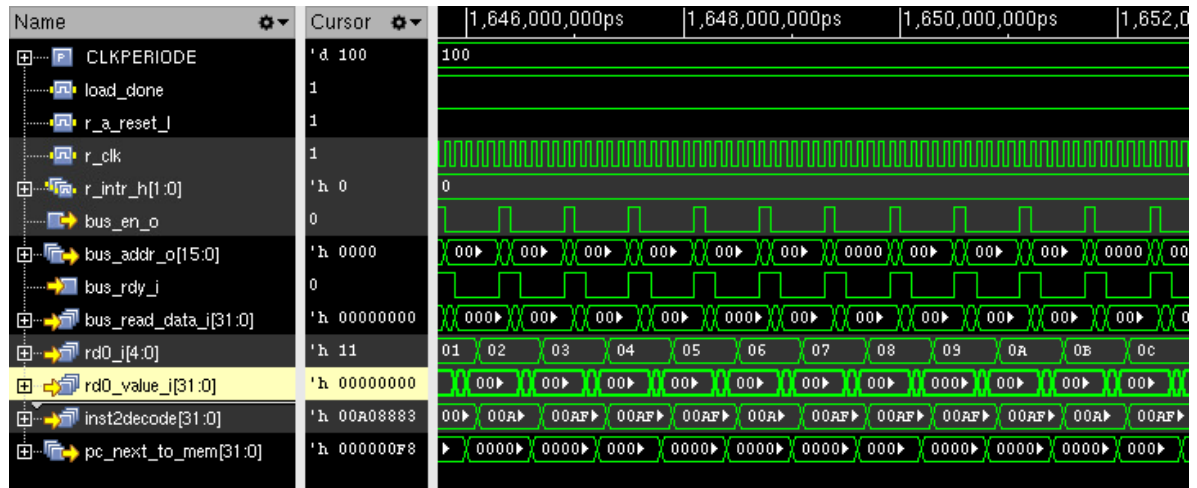


Figure 3.4: Loading from external memory

r0[31:0]	'h 00000000
r1[31:0]	'h 00000035
r2[31:0]	'h 00000035
r3[31:0]	'h 00000035
r4[31:0]	'h 00000035
r5[31:0]	'h 00004635
r6[31:0]	'h 00004635
r7[31:0]	'h 00004635
r8[31:0]	'h 00004635
r9[31:0]	'h 00004635
r10[31:0]	'h 00004635
r11[31:0]	'h 00004635
r12[31:0]	'h 00004635
r13[31:0]	'h 00000035
r14[31:0]	'h 00000001
r15[31:0]	'h 00000035
r16[31:0]	'h 00000035
r17[31:0]	'h 68574635
r18[31:0]	'h 68574635
r19[31:0]	'h 68574635
r20[31:0]	'h 68574635
r21[31:0]	'h 00000001
r22[31:0]	'h 00000001

Figure 3.5: Loading byte, word, half-word into regfile

[Figure 3.5](#) shows that the functionality of loading words, half-words and bytes is working properly.

Load from PRAM

Now, loading of words, half-words and bytes is also tested from the PRAM. Figure 3.6 shows this interaction and it can be seen that the instructions take a total of 4 clock cycles, two to load the instruction and the next two to load the data.

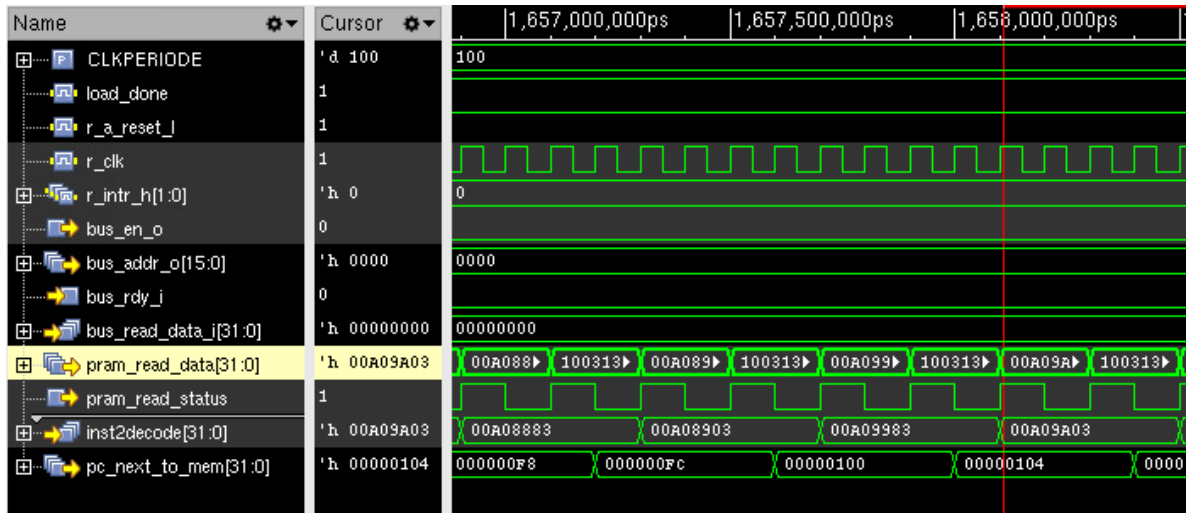


Figure 3.6: Loading words from PRAM

Single cycle instructions

All instructions that complete in one clock cycle within the core are now tested with the values in the regfile. These instructions comprise of:

- Immediate instructions like lui, auipc
- Logical instructions like and, or, not, xor, etc.
- Compare and shift instructions
- Arithmetic operations like add, sub, mul, div

Demonstrating the functionality of all these instructions is not practical for the sake of the report, but an example overview of these instructions is shown in Figure 3.7, with the corresponding changes reflected in the regfile registers.

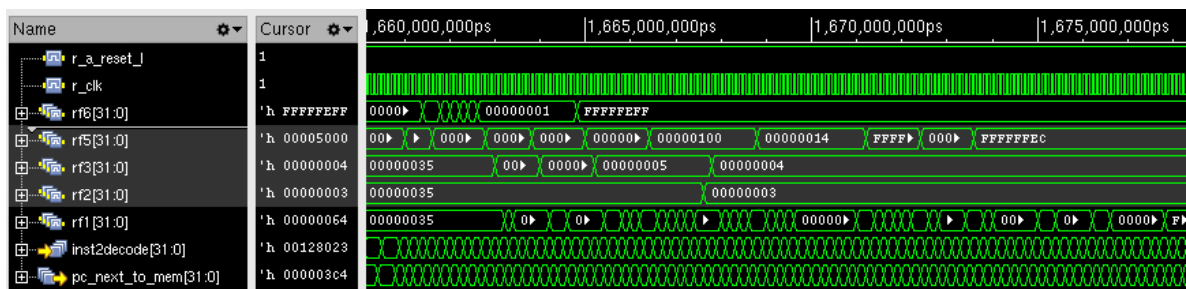


Figure 3.7: Series of single cycle instructions

Jump & Branch instructions

Since jump and branch instructions only require PC updation, they also complete in a single cycle inside the core. The main difference is that jump instructions store the next PC value from the current location to return to if the jump block finishes. This can be seen in [Figure 3.8](#) where the register x1 stores the PC when the jump instruction arrives and the PC returns to this value when the ret instruction is read.

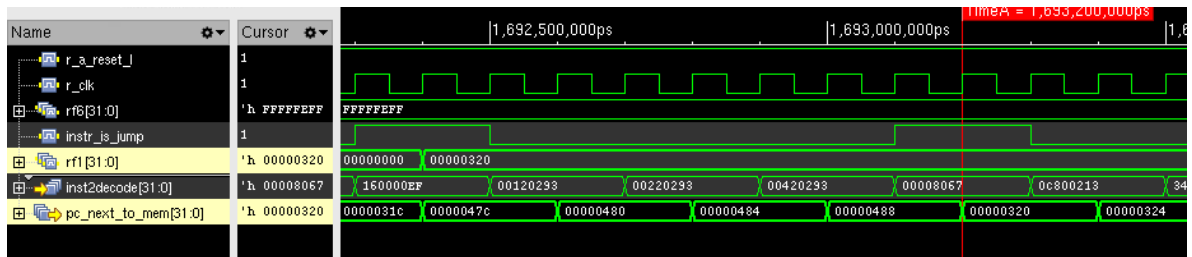
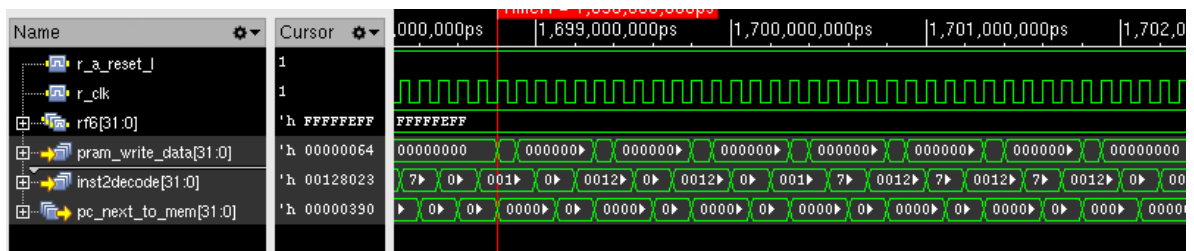


Figure 3.8: PC updation in Jump instructions

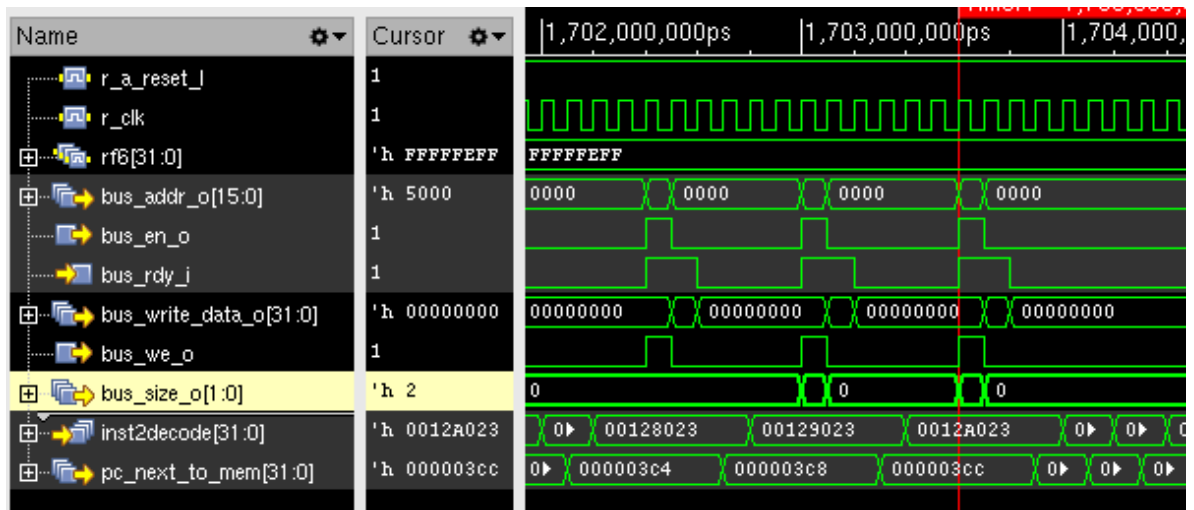
Store Instructions

Now the store operations are tested and verified. As with the load instructions, stores are verified for full word, half word as well as byte. [Figure 3.9](#) shows the behavior of store operations both to the PRAM as well as external memory.

It can be noticed that external load and store take the same amount of time to execute, but PRAM stores are one cycle faster than PRAM load instructions.



(a) PRAM Store



(b) External Store

Figure 3.9: Store operations to PRAM and EXT

CSR Instructions

Figure 3.10 shows the execution of the CSR atomic read & write instructions. Only the three registers mstatus, mepc and mcause are probed as misa and mtvec registers are hardwired. Highlighted by the pointer location is the output of the misa register which represents the ISA rv32IMC.

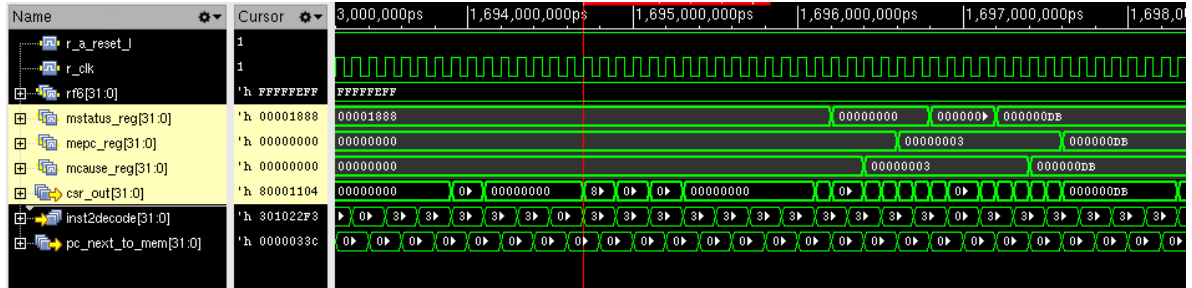


Figure 3.10: CSR Atomic instructions

System Instructions

Finally, the system instructions are verified. Here, the instructions ECALL/EBREAK, MRET and WFI are shown in Figure 3.11. First, the ecall instruction causes PC to jump to 0x10, while saving the previous PC value in MEPC register. Then, according to the specification in the ASM code, the PC jumps to a default handler which holds the MRET instruction. The MRET instruction now resets the MSTATUS register and causes the PC to load back to its previous value from MEPC.

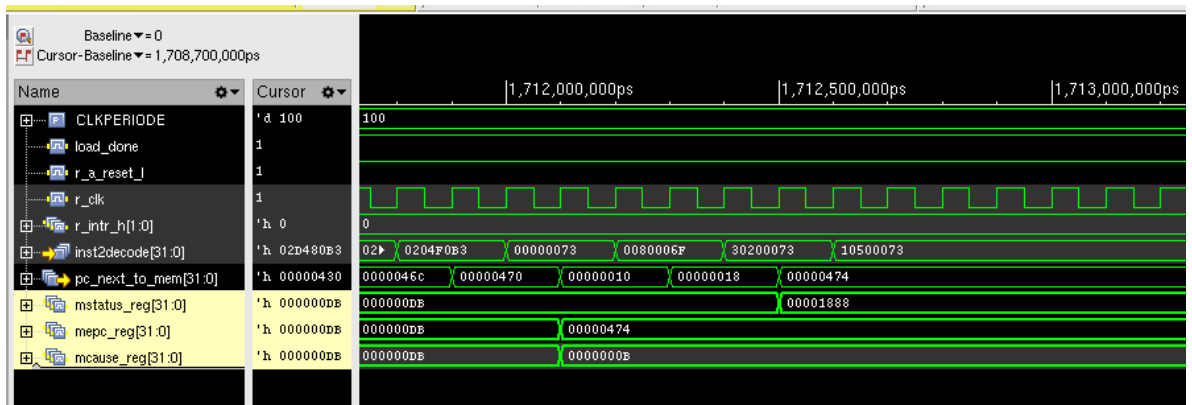


Figure 3.11: Waveforms of System instructions

Followed by this instruction sequence is the WFI, which puts the system into an infinite loop till an interrupt is received. An interrupt is now provided to the core, causing the PC to once again go into the interrupt handler, as can be seen in Figure 3.12.

It should be noted that the interrupt disables the MIE bit of the MSTATUS register, disabling acknowledgement of further interrupts till the PC breaks out of the current handler. This

is done again by the MRET instruction which loads the PC back to its previous value for continuing operations.

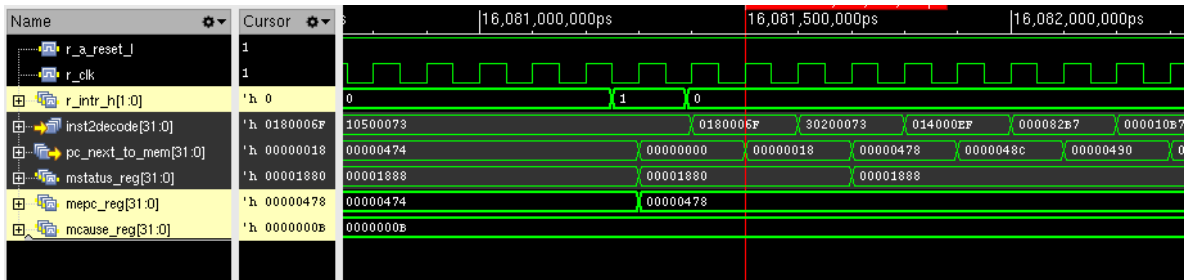


Figure 3.12: Requesting an Interrupt during WFI

3.3.2 RV32IMC ISA with C codes

In the current implementation, compressed instructions are decoded by expanding into a normal instruction first. Hence, the coverage report for the module already guarantees functionality.

In this section, this is verified by running a list of C codes compiled into RISC-V program codes. These codes are run both with and without compressed instructions enabled in the makefile for comparison.

Below are the C codes used and the results of running them on the Dobby SoC. The return value of the main function in these C codes always writes back to memory location 0x2000. Since Cadence NCSim does not naturally allow probing of register arrays, the result is probed from the location of the data in the register file, specifically at register x10.

factorial.c

The code in [Figure 3.13](#) calculates the factorial of 10 and returns the value to memory.

```
1 #include <corelib.h>
2 int main ()
3 {
4
5     int n, i;
6     unsigned long long fact = 1;
7
8     n = 10;
9     if (n < 0)
10        return fact;
11    else {
12        for (i = 1; i <= n; ++i) {
13            fact *= i;
14        }
15    }
16
17    return fact;
18
19
20 }
```

Figure 3.13: factorial.c

Expected value: 3628800 = 0x375f00

Register x10 in uncompressed program code: 00375f00

Register x10 in compressed program code: 00375f00

bubble.c

The code in [Figure 3.14](#) sorts the 5 number 1-5 and stores them back in memory. The return value of the main file is the first element in the sorted array.

```
1 #include <corelib.h>
2
3
4 int main ()
5 {
6     int arr[5];
7     arr[0] = 3;
8     arr[1] = 5;
9     arr[2] = 1;
10    arr[3] = 2;
11    arr[4] = 4;
12
13    int tmp;
14
15    for(int i = 0; i < 4; i++){
16        for(int j = 0; j < 4 - i; j++){
17            if(arr[j] > arr[j + 1]){
18                tmp = arr[j];
19                arr[j] = arr[j + 1];
20                arr[j + 1] = tmp;
21            }
22        }
23    }
24    return arr[0];
25 }
```

Figure 3.14: bubble.c

Expected value: 1 = 0x1

Register x10 in uncompressed program code: 00000001

Register x10 in compressed program code: 00000001

armstrong.c

The code in [Figure 3.15](#) finds the first armstrong number in the range 2-200 and returns that value.

```
1 #include<corelib>
2
3 int main()
4 {
5     int n, sum, i, t, a;
6
7     for(i = 2; i <= 200; i++)
8     {
9         t = i;
10        sum = 0;
11        while(t != 0)
12        {
13            a = t%10;
14            sum += a*a*a;
15            t = t/10;
16        }
17
18        if(sum == i)
19            i = 200;
20    }
21
22    return sum;
23 }
```

Figure 3.15: armstrong.c

Expected value: 153 = 0x99

Register x10 in uncompressed program code: 00000099

Register x10 in compressed program code: 00000099

matmul.c

The code in [Figure 3.16](#) performs dot product between two 3x3 matrices and returns the result.

```
1 #include <corelib.h>
2
3 int dotProduct(int matrix1[][3], int matrix2[][3], int rows, int cols) {
4     int result = 0;
5
6     for (int i = 0; i < rows; i++) {
7         for (int j = 0; j < cols; j++) {
8             result += matrix1[i][j] * matrix2[i][j];
9         }
10    }
11
12    return result;
13 }
14
15 int main() {
16     int matrix1[3][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
17     int matrix2[3][3] = {{9, 8, 7}, {6, 5, 4}, {3, 2, 1}};
18     int rows = 3;
19     int cols = 3;
20
21     return dotProduct(matrix1, matrix2, rows, cols);
22 }
23
24
```

Figure 3.16: matmul.c

Expected value: 165 = 0xa5

Register x10 in uncompressed program code: 000000a5

Register x10 in compressed program code: 000000a5

4 Synthesis and Physical Design

4.1 Synthesizability check of Dobby SoC

Cadence HAL RTL checker has been utilized to eliminate errors and warnings. It is imperative that there are no HDL errors or warnings that could impact the design's functionality. As illustrated in [Figure 4.1](#), the proposed design has been verified, and no errors were detected. Moreover, all critical warnings have been addressed and resolved.

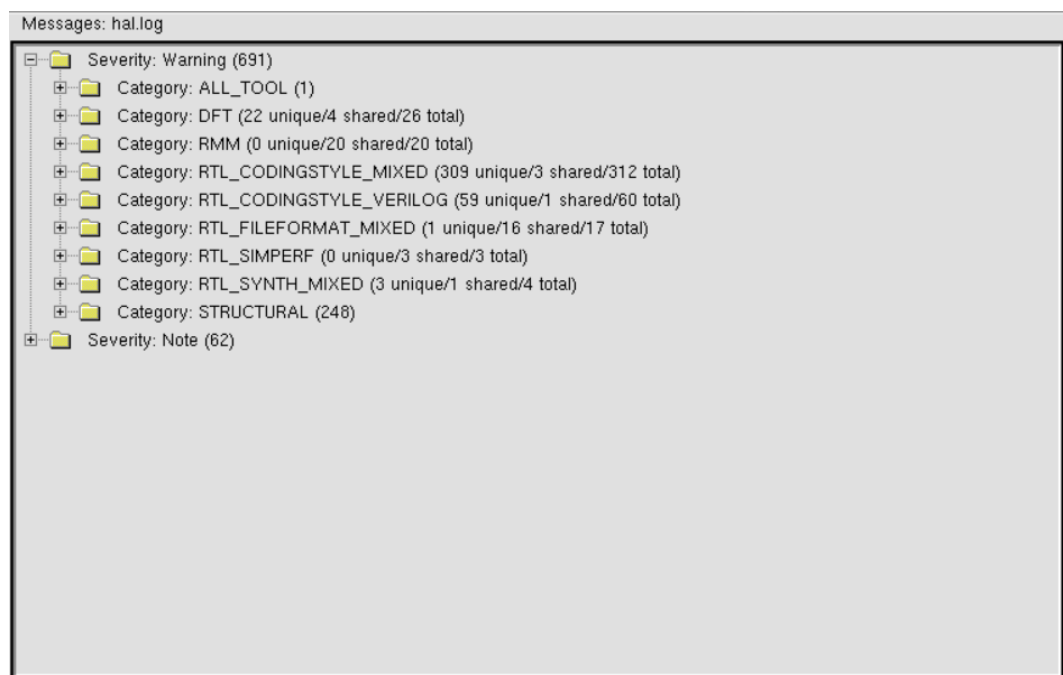


Figure 4.1: Depiction of HAL check log for the module doobby_soc

During HAL checks in the Cadence tool, a variety of warnings were encountered that correspond to different potential issues in the design. The warnings listed, ranging from DALIAS, FDTHRU, and URDWIR represent diverse concerns in the design process. However, not all warnings necessarily indicate critical issues. Some warnings such as DALIAS are related to naming conventions or aliases which, while worth noting, do not typically impact the functionality of the design. Others like MPCMPE and TPOUNR could pertain to comparisons or unconnected pins respectively, which in certain design contexts can be intentional. It is essential to understand the context of the design and the intention behind each module or

component. In some cases, these warnings can simply highlight unconventional methods or configurations that are still functionally correct. Therefore, while it is crucial to review and understand each warning, not all of them pose a threat to the design's reliability.

4.2 Synthesis Results

During logic synthesis, a high-level description of the circuit is converted to a technology-mapped netlist. It includes the following steps: parsing the HDL file, logic minimization, logic optimization, technology decomposition, and technology mapping. Synopsis Design Compiler tool has been employed to perform logic synthesis. The synthesis environment was setup by mapping "hpsnlib_g28_9t_RVT" (core logic library), "hpsnio_g28_HIO18" (IO pad library), and "hpsnmem_g28_HM_1P_GF28SLP" (SRAM macro library), setting "hpsnlib_g28_9t_RVT" as the target library. This is followed by specifying the worst-case, best-case, and typical conditions for synthesis.

```

1 #####
2 # definition of clocks
3 #####
4 create_clock -period 6.5 -waveform { 0 3.25 } [get_ports {I_CLK}] -name CLK
5 set_clock_transition 0.3 [get_clocks {CLK}]
6 set_clock_uncertainty 0.2 -setup [get_clocks {CLK}]
7 set_clock_latency 1.5 [get_clocks {CLK}]
8
9 #####
10 # Set I/O delays
11 #####
12 set_output_delay -clock [get_clocks {CLK}] 3.25 [all_outputs]
13 set_input_delay -clock [get_clocks {CLK}] 2 [remove_from_collection [all_inputs] [get_ports "I_CLK"]]
14
15 #####
16 # Set input, inout and output port conditions
17 #####
18 # Additional load values for all inputs
19
20 set_load 1.000 [all_inputs]
21
22 # Capacitance for all output or inout ports.
23 set_load 1.000 [all_outputs]
24
25 #
26 # Design attributes
27 #
28 set_dont_touch pads_i/i_HIO*
29 set_dont_touch pads_i/*_pad_*
30

```

Figure 4.2: Synthesis constraints specified in the constraints.tcl file

The synthesis constraints are specified in the "constraints.tcl" file as shown in [Figure 4.2](#). Parameters such as the target clock frequency, input and output delays, and clock latency are typically specified in this file.

Initially, the input and output delays are set to half of the targeted clock frequency with the aim of improving the timing results. After observing a few positive outcomes, hardware optimizations were done to reduce the critical path and to achieve a higher frequency as described in Chapter 2. Subsequently, input and output delay parameters were iteratively fine-tuned to enhance the results.

After completion of synthesis, the generated timing, area, and power reports are analyzed. The timing report includes the worst delay path (critical path) of 3 groups namely reg2reg, in2reg, and reg2out. The slack must be positive or else it would be considered as a timing

violation. The power and area report estimates the total power consumption and estimated area before routing of the design respectively.

Timing report analysis

The timing summary for the design, targeting a final clock period of 154 MHz using the "hpsnlib_g28_9t_RVT" library are illustrated in [Figure 4.3](#), [Figure 4.4](#), [Figure 4.5](#) below. It is evident that the timing is favorable, showing zero or positive slack across all three critical paths: reg2reg, in2reg, and reg2out. To ensure the correct functioning of the circuit, maintaining a positive or zero slack is of utmost importance.

Startpoint: doobby_soc_i/mc_dut/pram_inl/pram_addr_reg_reg_3 (rising edge-triggered flip-flop clocked by CLK)		
Endpoint: doobby_soc_i/core_dut/pc_unit_inst/program_counter_reg_11_ (rising edge-triggered flip-flop clocked by CLK)		
Path Group: CLK		
Path Type: max		
library setup time	-0.10	7.70
data required time		7.70

data required time		7.70
data arrival time		-7.70

slack (MET)		0.00

Figure 4.3: Slack (REG2REG)

The above [Figure 4.3](#) presents the register-to-register critical path along pram_addr_reg and program_counter_reg with a slack of zero.

Startpoint: I_BUS_RDY (input port clocked by CLK)		
Endpoint: dobbysoc_i/mc_dut/bus_controller_in/reg_write_data_reg_0_		
(rising edge-triggered flip-flop clocked by CLK)		
Path Group: REGIN		
Path Type: max		
Point	Incr	Path
clock CLK (rise edge)	0.00	0.00
clock network delay (ideal)	1.50	1.50
input external delay	2.00	3.50 f
I_BUS_RDY (in)	0.00	3.50 f
pads_i/I_BUS_RDY (pads)	0.00	3.50 f
pads_i/bus_rdy_pad_i/PAD (HI018_GF28SLP_IOPAD)	0.00	3.50 f
pads_i/bus_rdy_pad_i/DATA_IN (HI018_GF28SLP_IOPAD)	0.29	3.79 f
pads_i/bus_ready (pads)	0.00	3.79 f
dobbysoc_i/bus_rdy_i (dobbysoc)	0.00	3.79 f
dobbysoc_i/mc_dut/bus_rdy_i (top_memory_controller)	0.00	3.79 f
dobbysoc_i/mc_dut/bus_controller_in/bus_rdy_i (bus_controller)	0.00	3.79 f
dobbysoc_i/mc_dut/bus_controller_in/U7/Z (H_9T_RVT_INVX1)	0.05	3.84 r
dobbysoc_i/mc_dut/bus_controller_in/U6/Z (H_9T_RVT_NOR3X1)	0.08	3.92 f
dobbysoc_i/mc_dut/bus_controller_in/U5/Z (H_9T_RVT_NOR2X1)	0.16	4.08 r
dobbysoc_i/mc_dut/bus_controller_in/U10/Z (H_9T_RVT_NAND3X1)	0.23	4.30 f
dobbysoc_i/mc_dut/bus_controller_in/U133/Z (H_9T_RVT_A0I12X05)	0.65	4.96 r
dobbysoc_i/mc_dut/bus_controller_in/U9/Z (H_9T_RVT_BUFX2)	0.78	5.74 r
dobbysoc_i/mc_dut/bus_controller_in/U172/Z (H_9T_RVT_A0I22X05)	0.23	5.97 f
dobbysoc_i/mc_dut/bus_controller_in/U83/Z (H_9T_RVT_INVX1)	0.15	6.11 r
dobbysoc_i/mc_dut/bus_controller_in/reg_write_data_reg_0_/D (H_9T_RVT_DFPRQX1)	0.00	6.11 r
data arrival time		6.11
clock CLK (rise edge)	6.50	6.50
clock network delay (ideal)	1.50	8.00
clock uncertainty	-0.20	7.80
dobbysoc_i/mc_dut/bus_controller_in/reg_write_data_reg_0_/CP (H_9T_RVT_DFPRQX1)	0.00	7.80 r
library setup time	-0.13	7.67
data required time		7.67
data arrival time		-6.11
slack (MET)		1.55

Figure 4.4: Critical Path (REGIN)

The above Figure 4.4 presents the input-to-register critical path along the I_BUS_RDY pad and registers in the bus controller unit with a positive slack of 1.55 ns.

Startpoint: doobby_soc_i/mc_dut/bus_controller_in/reg_access_reg (rising edge-triggered flip-flop clocked by CLK)		
Endpoint: B_BUS_DATA[0] (output port clocked by CLK)		
Path Group: REGOUT		
Path Type: max		
Point	Incr	Path
clock CLK (rise edge)	0.00	0.00
clock network delay (ideal)	1.50	1.50
doobby_soc_i/mc_dut/bus_controller_in/reg_access_reg/CP (H_9T_RVT_DFPRQX1)	0.00	1.50 r
doobby_soc_i/mc_dut/bus_controller_in/reg_access_reg/Q (H_9T_RVT_DFPRQX1)	1.10	2.60 r
doobby_soc_i/mc_dut/bus_controller_in/bus_en_o (bus_controller)	0.00	2.60 r
doobby_soc_i/mc_dut/bus_en_o (top_memory_controller)	0.00	2.60 r
doobby_soc_i/bus_en_o (doobby_soc)	0.00	2.60 r
pads_i/bus_en (pads)	0.00	2.60 r
pads_i/U6/Z (H_9T_RVT_AND2X2)	0.40	2.99 r
pads_i/U7/Z (H_9T_RVT_BUFK20)	0.37	3.36 r
pads_i/bus_data_pad_i_0/PAD (HI018_GF28SLP_IOPAD)	1.06	4.42 f
pads_i/B_BUS_DATA[0] (pads)	0.00	4.42 f
B_BUS_DATA[0] (inout)	0.00	4.42 f
data arrival time		4.42
clock CLK (rise edge)	6.50	6.50
clock network delay (ideal)	1.50	8.00
clock uncertainty	-0.20	7.80
output external delay	-3.25	4.55
data required time		4.55
data required time		4.55
data arrival time		-4.42
slack (MET)		0.13

Figure 4.5: Critical Path (REGOUT)

The above Figure 4.5 presents the register-to-output critical path along an output register in the bus controller unit and output pad with a positive slack of 0.13 ns.

Power report analysis

It is evident that as the frequency rises, power consumption increases proportionally. This phenomenon arises from the increased switching activity within the circuit. Figure 4.6 provides an overview of the power analysis for the design, utilizing a clock of 154 MHz. The total power consumption of the SoC is 0.878 mW. It is noteworthy to mention that for previous unoptimized versions of the design, the power consumption was relatively lower.

Operating Conditions: wc_0d76V_m40C Library: hpsnlib_g28_9t_RVT_wc_0d76V_m40C
Wire Load Model Mode: top

Global Operating Voltage = 0.76
Power-specific unit information :
Voltage Units = 1V
Capacitance Units = 1.000000ff
Time Units = 1ns
Dynamic Power Units = 1uW (derived from V,C,T units)
Leakage Power Units = 1nW

Cell Internal Power = 831.0162 uW (95%)
Net Switching Power = 47.0539 uW (5%)

Total Dynamic Power = 878.0701 uW (100%)
Cell Leakage Power = 215.0299 nW

Power Group	Internal Power	Switching Power	Leakage Power	Total Power	(%)	Attrs
io_pad	564.8428	9.9714	119.4302	574.9333	(65.46%)	
memory	149.5892	1.0778	0.0000	150.6671	(17.15%)	
black_box	0.0000	0.0000	0.0000	0.0000	(0.00%)	
clock_network	6.1118	8.8159	0.5171	14.9282	(1.70%)	
register	93.8830	0.4705	24.4042	94.3779	(10.75%)	
sequential	0.1868	2.4610e-02	0.2890	0.2117	(0.02%)	
combinational	16.4024	26.6937	70.3773	43.1664	(4.91%)	
Total	831.0159 uW	47.0539 uW	215.0179 nW	878.2846 uW		

Figure 4.6: Power consumption report

Area report analysis

As the operating frequency increases, there is a corresponding rise in area utilization. This is likely due to the necessity for additional circuitry to support higher frequencies in order to meet the design's timing constraints. Achieving a stable and synchronized clock tree becomes increasingly challenging at higher frequencies, often requiring the incorporation of additional buffers along the clock path, further contributing to increased area utilization. Figure 4.7 presents the area analysis for the design with a clock period of 154 MHz. It highlights that the final design exhibits more leaf cells, cell area, and net area in comparison to the lower frequencies.

Library(s) Used:

hpsnlib_g28_9t_RVT_wc_0d76V_m40C (File: /home2/vlsi00/prz-root/ic
hpsnio_g28_HI018_wc_0d76V_1d62V_m40C (File: /home2/vlsi00/prz-root/ic
HM_1P_GF28SLP_wc_0d76V_m40C (File: /home2/vlsi00/prz-root/ic

Number of ports:	59
Number of nets:	150
Number of cells:	2
Number of combinational cells:	0
Number of sequential cells:	0
Number of macros/black boxes:	0
Number of buf/inv:	0
Number of references:	2
Combinational area:	11738.022931
Buf/Inv area:	1062.828018
Noncombinational area:	3949.335068
Macro/Black Box area:	336394.835938
Net Interconnect area:	undefined (No wire load specified)
Total cell area:	352082.193936
Total area:	undefined

Figure 4.7: Area report

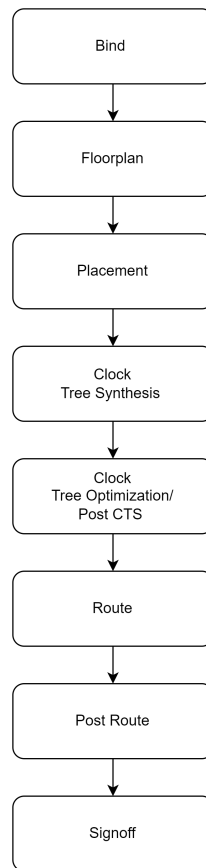


Figure 4.9: Flowchart of steps involved in the physical design phase

The actual physical implementation of the design involves various factors such as routing delays, wire lengths, capacitive loading, etc. These physical aspects significantly affect the maximum achievable frequency. Hence, for the physical design phase, the operating frequency of the SoC is selected to be 132 MHz. This facilitates relaxed placement and routing phases.

4.3.1 Bind

Importing the synthesis results, including the technology-mapped netlist and timing constraints, is a crucial step in the design process. Using the "icfe bind" command can seamlessly integrate these critical components into their design environment. This integration ensures that the synthesized netlist aligns with the specified timing constraints.

4.3.2 Floor Planning and Power Planning

Floor planning and power planning are integral stages in the chip design process. During floor planning, the chip's overall dimensions are determined, along with a tentative arrangement of units like [Intellectual Properties \(IPs\)](#) and macros. These units are strategically positioned at the required locations within the Core area, where logic cells reside, while the Die area

represents the chip's overall surface. To ensure optimal power distribution, power planning is conducted concurrently. This involves creating a power distribution network with a power ring, and connecting it to VDD and VSS pads. Power straps are introduced to minimize resistance-inductance effects and ensure that logical cells draw the necessary supply from nearby sources. Effective power planning results in reduced heat dissipation, minimized IR drops, and mitigates issues related to electron migration.

The position of IO pads is specified in the "dobby_soc.io" file along with the creation of power supply pads. The die area is specified as 1500 x 1500 μm^2 in the "parameters.tcl" file. An offset of 50 μm from core to pad is specified.

4.3.3 Placement

Once the power distribution network has been established, the standard cells are strategically arranged within the designated rows of the floorplan area. This arrangement process unfolds in two distinct phases: Global Placement and Detailed Placement. During the Global Placement phase (also referred to as Fast Placement), the tool prioritizes speed in positioning the cells, while striving to identify optimal locations for them. This phase is considered "soft" placement, as the cell blocks are not firmly locked into place. As a result, some minor overlaps or misalignments between rows may exist. Subsequently, the Detailed Placement stage takes over from the global placement process, rectifying any placement issues while adhering to the predefined rules established during global placement. The position of macros was specified. Existing scripts were utilized to perform this process.

4.3.4 Clock Tree Synthesis

One of the central objectives of any RTL design process is to minimize clock latency, ideally achieving a state of zero latency. This ensures that every register in the design receives the clock signal simultaneously and without any deviations. Any disparities in timing could lead to registers capturing data at disparate moments, potentially resulting in erroneous functionality. Consequently, there is a need to construct a clock distribution tree that efficiently routes the clock signal to all registers while minimizing skew to nearly zero. In this phase, clock buffers are introduced into the clock path to uphold signal integrity and enhance driving capability. Various algorithms, such as H Trees and X Trees, are employed to accomplish this task.

4.3.5 Routing

During this phase, the previously placed blocks are interconnected using physical wires. Up to this point, timing analysis has been based on an idealized scenario with zero delays assumed for all connections. Now, it becomes imperative to conduct timing analysis once more, taking into account the real wire delays to ensure there are no timing violations. When performing routing, all Design Rule Checks (DRC) are enabled to ensure proper routing, guaranteeing that the design can be fabricated without encountering any issues. Additionally, a range of other critical checks come into play, including Signal Integrity, Cross-talk analysis, Detailed Power Analysis, and the insertion of diodes to mitigate the Antenna Effect associated with

metal components. These extensive checks are essential in modern chip design to ensure the highest level of performance and reliability.

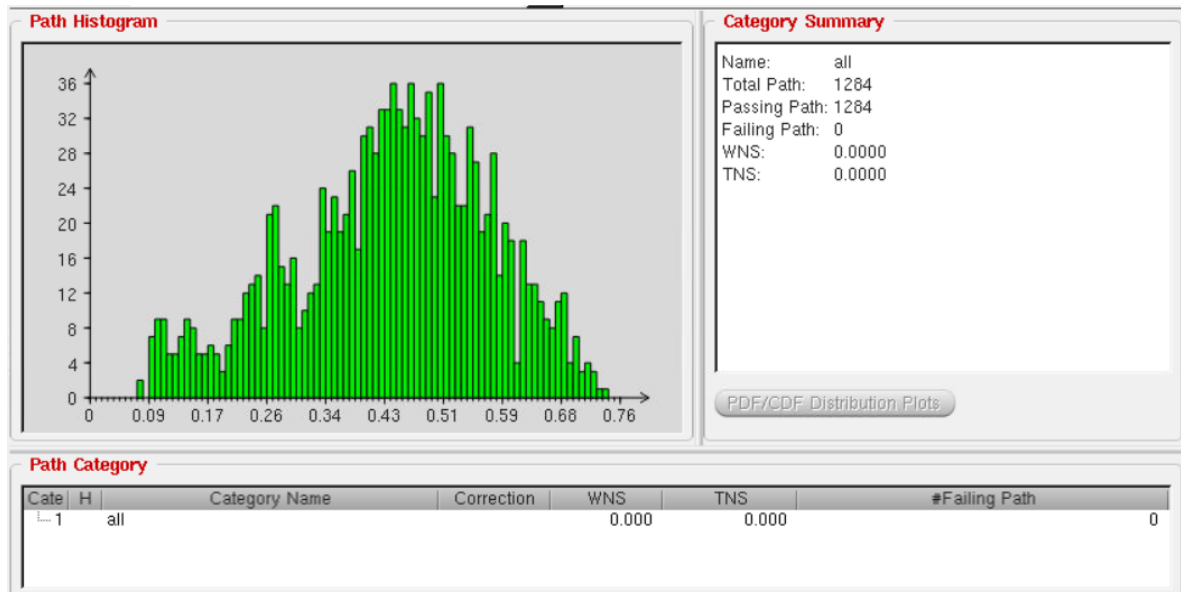


Figure 4.10: PostRoute setup timing Histogram

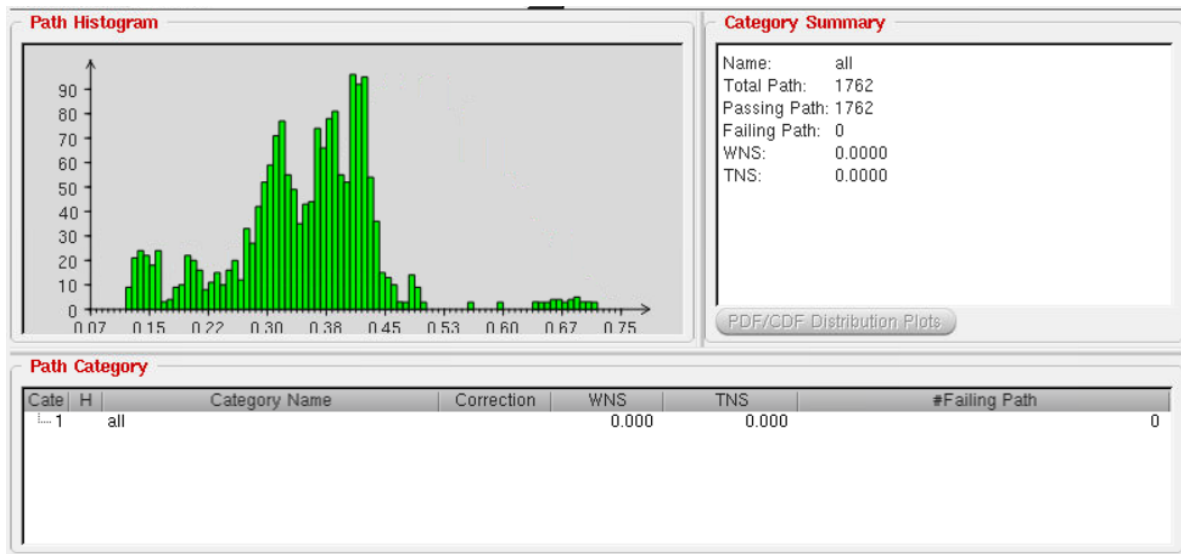


Figure 4.11: PostRoute Hold timing Histogram

Figure 4.10 and Figure 4.11 depict the path histograms for setup and hold slack after the physical design process. These histograms are a visual representation of routing congestion with respect to setup and hold requirements. It also represents slack distribution along the critical path, which in this case indicates the absence of timing violations.

4.3.6 Signoff

This step adds filler cells in the gaps of standard cells and performs a final check on unrouted nets, DRC, signal integrity, and timing. Then, the design is ready to be exported to a GDSII file for tape-out.

Figure 4.12 depicts the final layout of the chip after the physical design phase.

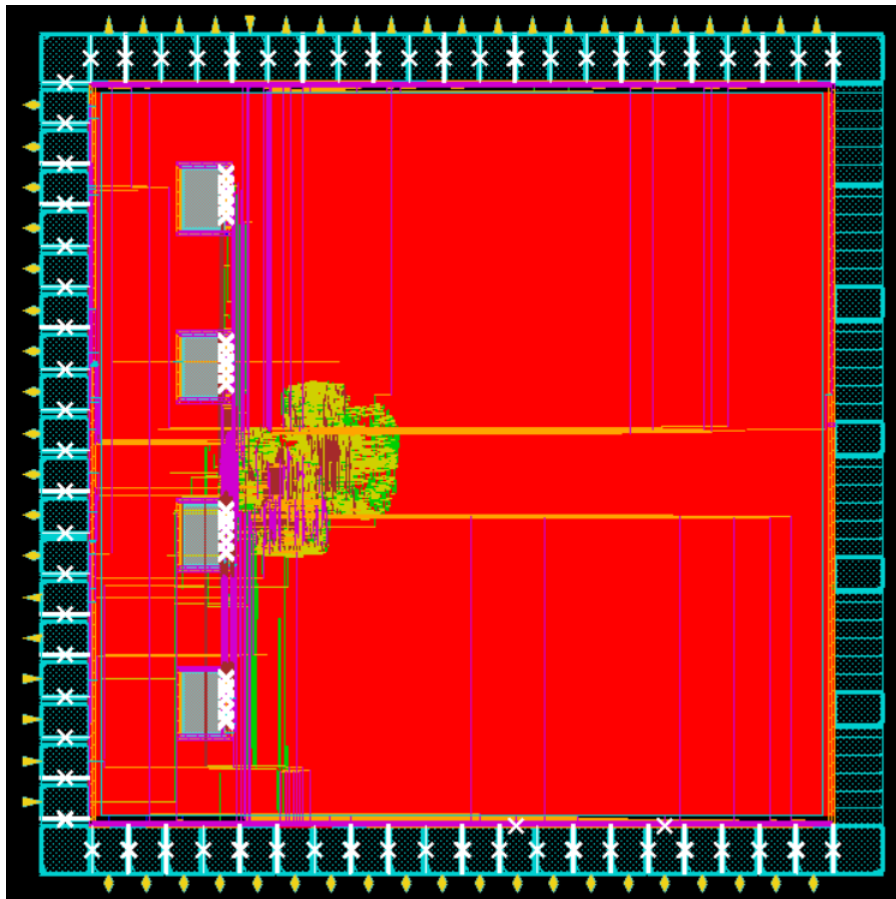


Figure 4.12: Chip layout

5 Conclusion

A RISC-V based SoC meeting the presented task specifications has been designed in RTL. The SoC has been implemented using the GF28nm technology library. The developed SoC targeting performance can operate at a maximum frequency of around 150 MHz. Design decisions and efforts made to achieve this frequency have been presented. Resulting tradeoffs in area utilization and power consumption have been studied. Individual components within the SoC have been verified to achieve a high simulation coverage. The verification results of the Dobby SoC have been analyzed in detail to enhance the reliability of the design.

There are several promising paths for future development and expansion in the current RISC-V design. One notable approach involves the integration of additional RISC-V extensions to extend the processor's functionality. These extensions including the F(Floating-Point), D(Double Precision Floating-Point), and V(Vector) extensions have the potential to enlarge the processor's utility, especially in scientific and multimedia computing applications.

Additionally, transitioning from the current architecture to a more complex pipelined design offers performance enhancement. A pipelined approach can enable concurrent execution of instructions, potentially leading to improved clock frequencies and throughput. Furthermore, the implementation of advanced out-of-order execution techniques will help boost instruction-level parallelism, ultimately resulting in more efficient processing.

6 References

- [1] The RISC-V Instruction Set Manual Volume I: User-Level ISA Document Version 2.2, University of California, Berkeley
- [2] The 8051 Microcontroller and Embedded Systems Using Assembly and C, Second Edition, Muhammad Ali Mazidi, Janice Gillispie Mazidi, Rolin D. McKinlay.
- [3] Computer Architecture: A Quantitative Approach, Fifth Edition, John L. Hennessy and David A. Patterson.
- [3] An Overview of RISC vs CISC, Alan D. George, Department of Electrical Engineering FA-MUFSU College of Engineering Tallahassee, FL 32304.
- [5] Digital Design and Computer Architecture, second Edition, David Money Harris and Sarah L. Harris.
- [6] The RISC-V Reader: An Open Architecture Atlas, third edition, David A. Patterson and Andrew Waterman.
- [7] Cadence Design Systems, Inc. Cadence Hal. Cadence Design Systems, Inc.
- [8] Cadence Design Systems, Inc. Cadence NC-Sim. Cadence Design Systems, Inc.
- [9] Synopsys, Inc. Synopsys Design Compiler. Synopsys, Inc.
- [10] Cadence Design Systems, Inc. Cadence Innovus. Cadence Design Systems, Inc.

A Appendix

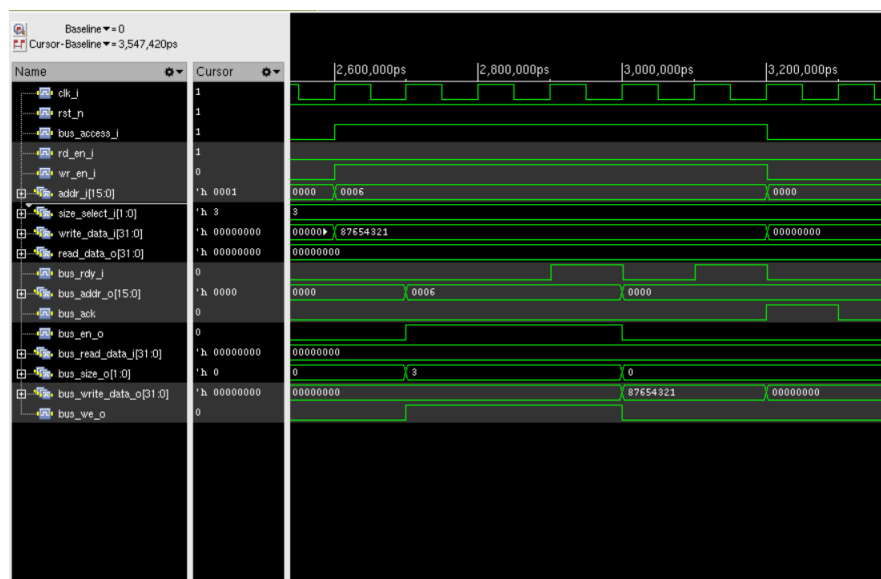


Figure A.1: Bus controller handling wait cycles during a bus write operation

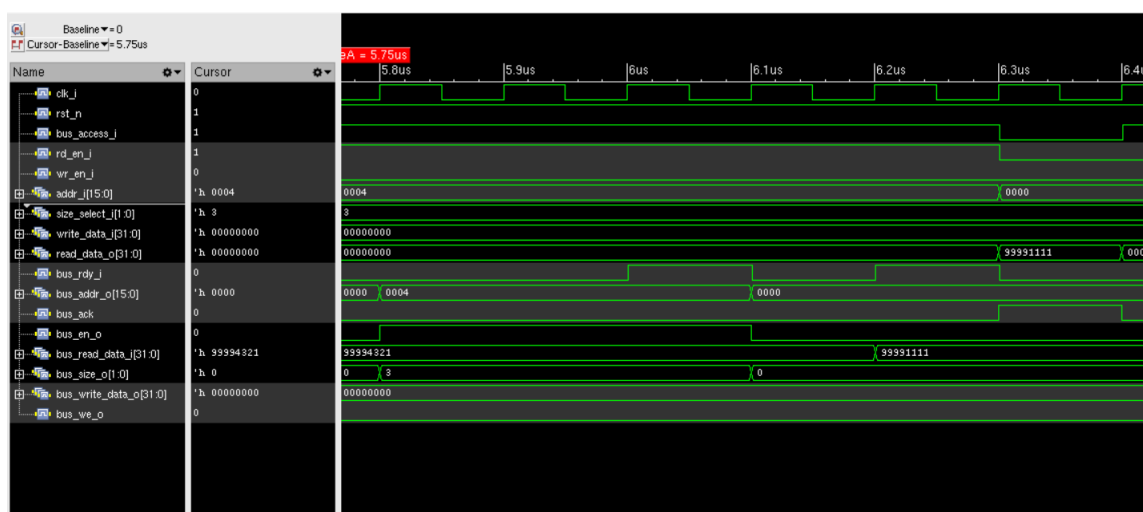


Figure A.2: Bus controller handling wait cycles during a bus read operation