2023

Custom Processor Architecture

DESIGN AND PRE-SILICON VERIFICATION OF A SIMPLE PROCESSOR

TOM DIEDEREN

Revision History

Date	Version	Comments
Aug 2023	0.1	Added high-level hardware and testbench architecture.
Sept 2023	0.3	Hardware and unit testbench architecture definition completed.
Oct 2023	0.5	Hardware RTL written, unit tests completed.
Nov 2023	0.7	Architecture of UVM system level testbench completed.
Dec 2023	0.8	UVM tests completed.
Jan 2024	1.0	Included lessons learned and future improvements.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

Table 1 Acronyms used in this document.

Acronym	Meaning	Description
ALU	Arithmetic Logic Unit	Digital circuit that can perform arithmetic and logic operations on binary integers.
BCD	Binary Coded Decimal	Number representation format which uses 4 bits per base 10 digit.
DUT	Device Under Test	The design being verified with the testbench.
FPGA	Field Programmable Gate Array	Circuit containing reconfigurable logic which allows for redesigning/updating hardware after point of sale (“in the field”).
HDL	Hardware Description Layer	Code in this layer represents hardware (synthesizable)
HVL	Hardware Verification Layer	Code in this layer is used for verification (not synthesizable)
I/O	Input / Output	Direction of Signals
IC	Integrated Circuit	An electronic circuit on a single piece of Si (also informally called “chip”)
OOP	Object Oriented Programming	Programming paradigm.
PCB	Printed Circuit Board	Printed Interconnect for electrical components.
Pre-Si	Pre-Silicon	Refers to the timeframe before the actual Silicon, that contains the design, is available.
ROM	Read-Only Memory	A type of memory that cannot be written to.
RTL	Register Transfer Layer	An abstraction layer of digital or mixed signal design.
Si	Silicon	Semiconductor. Element 14 on the periodic table.
TCL	Tool Command Language	Programming Language
UVM	Universal Verification Method	A standardized framework of SW classes enabling more reuse and standardized design verification.

SUMMARY

This project covered the design of a basic, yet fully functional, processor. The goals of this project were:

- Specify the micro-architecture for the processor.
- Design the RTL implementation of this micro-architecture (including implementation in Verilog).
- Design the system-level testbench architecture (using the UVM/SystemVerilog).
- Verify functional behavior pre-Si (simulation).

The project goals were met and resulted in increased (System) Verilog proficiency, greater familiarity with the UVM, as well as enhanced knowledge of computer architecture for the author.

Although the general, high-level, architecture is based on chapter 9 of [1], the detailed, low-level, microarchitecture, RTL design and Verilog implementation, UVM testbench architecture and SystemVerilog implementation are all newly created by the author and are not part of [1] or any other source at the time of writing. The author wrote this complete document without support from a large language model / AI.

Keywords/Skills:

Verilog, SystemVerilog, UVM, Micro-Architecture, Functional Verification, Hardware Design, Logic Design, Python, Simulation, Questa Sim, Project Management.

A note on readability: this document contains links to sections within it. Readability might be enhanced when read electronically.

1 Requirements & Specifications

This section contains a basic description of what the design should adhere to as well as some very simple project management information.

1.1 Project Outline

This project is for educational purposes and will not lead to a physical product. The project will create design "IP" which could be implemented in actual Si (e.g., in an FPGA). Doing so would again be mostly educational in nature since the performance of the design is probably not sufficient for commercial applications. Some suggestions that would enhance the performance to a potentially commercially viable level are made in the section on This project was purely educational. The ISA and micro-architecture were kept simple to make the scope manageable and compress the schedule of the project to one to two quarters of evening and weekend work. This section describes several enhancements that would improve the processor's performance as well as lessons learned during this project.

Potential Design Enhancements on page 97.

1.1.1 Project Scope

Only front-end design is in scope, meaning the micro-architecture and RTL design. Pre-Si verification will also be done. However, back end-design such as die size, gate count, power consumption, etc. will not be in scope.

1.1.2 Project Timeline and Budget

The allotted time for the project is 1 to 2 quarters of "after-work" hours (evenings and weekends). The budget is: $0.

1.2 CPU

The processor contains two main parts: the datapath and the control unit.

1.2.1 Datapath

The datapath contains an execution unit, EU, register file and three busses.

1.2.1.1 The Execution Unit

The EU performs non-signed, integer, arithmetic, bitwise logic operations, and shift operations. A complete list of supported CPU instructions is presented in the section:

Instructions on page 14. The EU will support the following micro-operations:

  - Add or Subtract the content of two registers.
  - Increment or decrement the content of a register.
  - Shift the content of a register right or left.
  - Bitwise AND
  - Bitwise OR
  - Bitwise XOR
  - Invert (not / 1's complement)
- HW RTL shall be described in Verilog
- 16-bit bus width in Verilog (parameterized for re-use)
- A zero status bit will be provided for arithmetic.
- Only unsigned arithmetic will be supported.

1.2.1.2 Busses

The design contains 3, 16-bit, busses: bus A, bus B, and bus D (whose nomenclature is a remnant of the high-level architecture [1]). Bus A is used for address output to RAM, Data Memory, as well as jump addresses. Bus B is used for data output to RAM. Bus D for data write back to a register file in the datapath.

1.2.2 Control Unit

The control unit contains the Program Counter, Instruction Register, and Instruction Decoder. This processor will have a simple architecture to make the scope and schedule manageable. The design is:

- Non-pipelined
- Scalar
- Has no interrupts
- No power management (on/off only)

1.2.2.1 Instruction memory

The Instruction Memory is asynchronous and read-only to facilitate single cycle execution (fetch, decode, execute). The instruction memory will hold a single sample program that executes all instructions once.

1.3 VERIFICATION

Functional verification will only be done Pre-Si and in two stages:

1. At the unit level by using custom testbenches written in Verilog.
2. At the system level through a UVM based testbench (SystemVerilog).

Post-Si verification, as well as Formal verification is out of scope for this project. Design and simulation will be down without power aware options (UPF). The HW has just two power states: on and off. No sleep or low power states were implemented.

The verification architecture will re-use some elements of the testbench architecture from one of the author's previous projects: [2].

1.3.1 EDA Tooling

Questa Sim, the free Intel FPGA version, will be used as EDA tool. Synopsis VCS trough edaplayground.com may have some additional functionality although the online user interface is less user-friendly for projects of this size (20+ source / testbench files). Unfortunately, the free Questa Sim version has some serious drawbacks:

- Constraint random transaction input is not supported.
- Cover groups are not supported.
- The number of sequence items per sequence is limited.

For a free version, these limitations make sense as they make the tool unusable for any “actual” verification work. A simple scoreboard will be built to verify the correctness of the CPU instructions. Even-though unsupported in the free version, a functional coverage checker will be built, and the driver sequence items will have randomizable fields to adhere to standard practice.

2 Instruction Set Architecture

To keep the scope, and schedule, of this project manageable, a simple ISA was chosen. It came with the high-level reference architecture that was further developed upon for this project [1].

2.1.1 Registers

The design contains the following 16-bit registers and memory:

- Eight, 16-bit, registers within a register file
- A 16-bit Program Counter
- Data memory, RAM (out of scope due to time limitations and will be simulated)
- Instruction memory, ROM,

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
General Purpose Register															
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

Figure 1 The design contains eight internal, 16-bit, registers.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Program Counter															
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

Figure 2 16-bit program counter

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Data Memory (RAM)															
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

Figure 3 The data RAM

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Instruction Memory (ROM)															
R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R

Figure 4 Instructions are fetched from ROM.

Since this project involves front-end design only, no capacity limit will be specified for the RAM and ROM.

2.1.2 Instruction Formats

There will only be three instruction formats for this design: register, immediate, and jump / branch.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Opcode							Destination Reg. (rd)			Source Reg. A (rsA)			Source Reg. B (rsB)		

Figure 5 Register Instruction Format (R-type)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Opcode							Destination Reg. (rd)			Source Reg. A (rsA)			imm[2:0]		

Figure 6 Immediate Instruction Format (I-type)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Opcode							Address Left (AD)			Source Reg. A (rsA)			Address Right (AD)		

Figure 7 Jump / Branch Instruction Format

To limit the scope, the register file will only contain 8 registers. This is sufficient to demonstrate functionality but most likely inadequate for any practical applications.

2.1.3 Instructions

The architecture supports 18 basic instructions. The 7 MSBs of the op code define the type of instruction. From these 7 bits, the 3 MSBs define the overall type, the 4 LSBs the sub-type.

Table 2 Instruction Specification of the Design (EU: Execution Unit)

Instruction	**Opcode**	**Mnemonic**	**Type**	**Format**	**Description**
Move A	000_0000	MOVA	EU with register(s)	R-type	rd <- rsA
Increment	000_0001	INC	EU with register(s)	I-type	rd <- rsA + 1
Add	000_0010	ADD	EU with register(s)	R-type	rd <- rsA + rsB
Subtract	000_0101	SUB	EU with register(s)	R-type	rd <- rsA - rsB
Decrement	000_0110	DEC	EU with register(s)	I-type	rd <- rsA - 1
AND	000_1000	AND	EU with register(s)	R-type	rd <- rsA ^ rsB
OR	000_1001	OR	EU with register(s)	R-type	rd <- rsA v rsB
XOR	000_1010	XOR	EU with register(s)	R-type	rd <- rsA ⊕ rsB
NOT	000_1011	NOT	EU with register(s)	R-type	rd <- $\overline{\text{rsA}}$
Move B	000_1100	MOVB	EU with register(s)	R-type	rd <- rsB
Shift Right	000_1101	SHR	EU with register(s)	R-type	rd <- shift right rsB
Shift Left	000_1110	SHL	EU with register(s)	R-type	rd <- shift left rsB
Load	001_0000	LD	Mem Read	R-type	rd <- RAM[rsA]
Store	010_0000	ST	Mem Write	R-type	RAM[rsA] <- rsB
Add Immediate	100_0010	ADI	EU with constant	I-type	rd <- rsA + zero fill imm[2:0]
Load Immediate	100_1100	LDI	EU with constant	I-type	rd <- zero fill imm[2:0]

Branch on Zero	110_0000	BRZ	Branch	Jump/Branch	if rsA == 0: PC <- PC + AD
					if rsA != 0: PC < PC + 1

Jump	111_0000	JMP	Jump	Jump/Branch	PC <- rsA

3 MICROARCHITECTURE

This chapter describes the micro-architecture of the processor. A rough, high-level, version can be found in: [1]. As part of this project, this high-level version was further elaborated into a complete, low-level, version ready for RTL implementation which is described in the next chapter.

3.1 TOP LEVEL

The top-level microarchitecture can be divided in two main parts: the datapath and control unit.

3.1.1 Datapath

The datapath contains a register file, with eight, 16-bit, register that can be read and written to, and an execution unit that can perform unsigned arithmetic, logic operations, and shift operations. The register file and execution unit are connected by 2 busses. Bus A is used for addresses and is connected to the Program Counter. Bus B is used for data and connected to external RAM.

The register file contains inputs that select 2 source registers, source register A, rsA, and source register B, rsB. The result of an operation will be written back to the destination register: rd. A register write enable, regWrite, signal enables writing to the register file.

The design contains two multiplexers, Mux B and Mux D, whose nomenclature is a remnant of the high-level architecture [1]. Mux B enables load and store immediate instructions, and mux D enables load (from RAM) instructions.

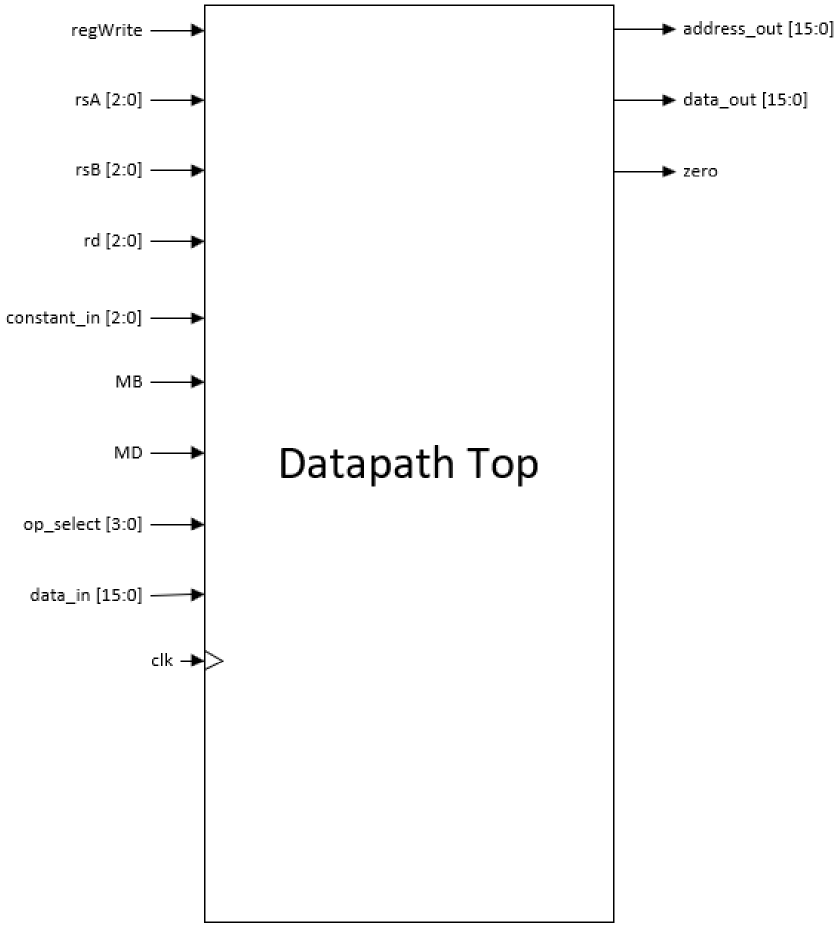


Figure 8 Top Level Overview of the Datapath

Table 3 Top-level Datapath I/O

Signal Label	**Full Name**	**I/O**	**Explanation**
regWrite	Register Write	I	High enables writing to registers
rsA[2:0]	Register, source, A	I	Register that holds operand for the EU
rsB[2:0]	Register, source, B	I	Register that holds operand for the EU
Rd [2:0]	Register, destination	I	Register in which EU op result will be stored
Constant_in[2:0]		I	Used for Add Immediate instruction
MB	Multiplexer (MUX) B	I	Selects Mux B input: const_in or reg_file Out
MD	MUX D	I	Selects Mux D input: EU out or data_in
Op_select [3:0]	Operation Select	I	Selects which operation the EU performs
Data_in[15:0]		I	Input from RAM
clk	Clock	I	Main clock signal
Address_out[15:0]		O	Connected to Bus A. Used for Jumps, RAM Wr.
Data_out[15:0]		O	Output to RAM
zero		O	ALU Status Bit (not used in this design)

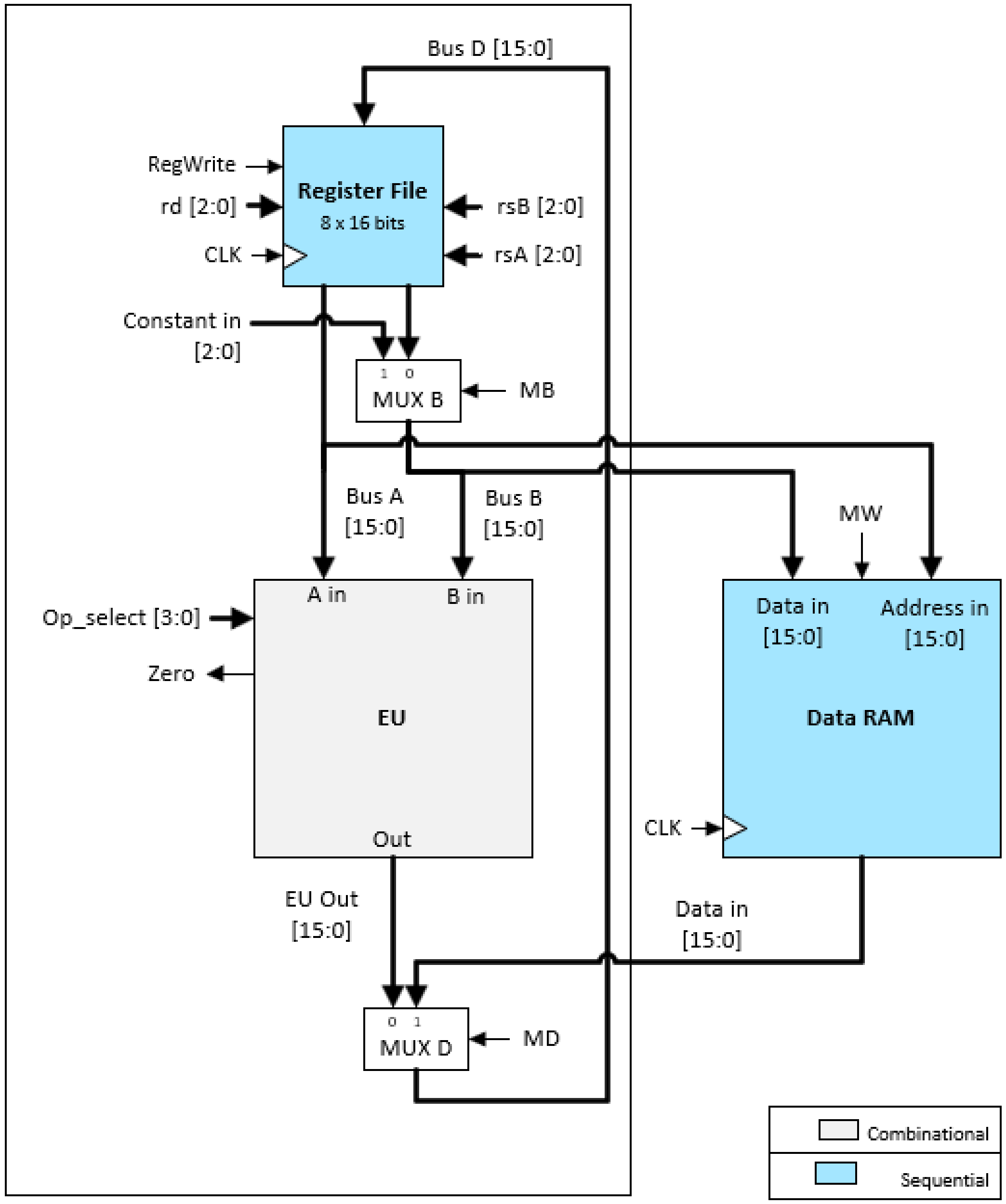


Figure 9 Detailed Overview of the Datapath

3.1.1.1 Execution Unit (EU)

The execution unit performs arithmetic on unsigned binary input, as well as execute logic and shift operations. An input called op_select specifies which operation is to be performed. 0xxx is the op_select value for arithmetic, 10xx is used for logic operations, and 11xx for shifts. These values match instruction bits [12:9] as can be seen in Table 2 on page 14. The EU does have a zero-status bit although it is not used in this design.

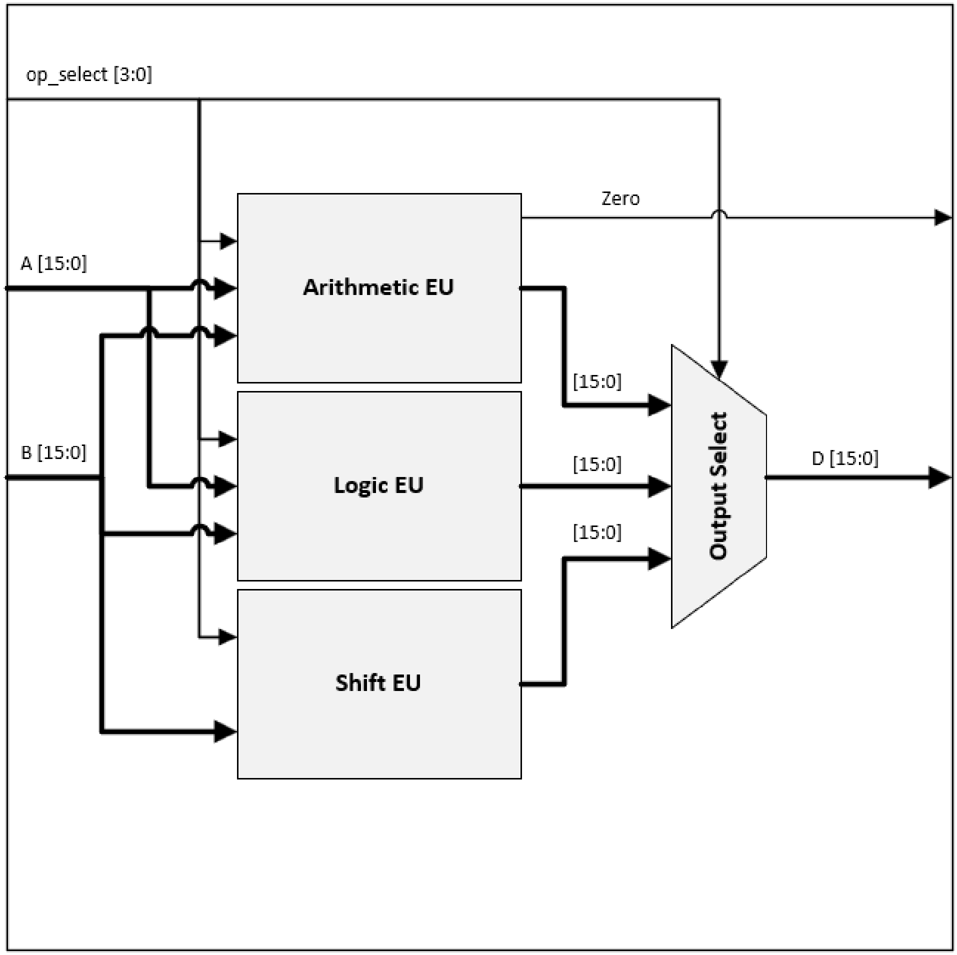


Figure 10 Architecture of the Execution Unit

Signal Label	Full Name	I/O	Explanation
op_select [3:0]	Operation Select	I	Selects which operation the EU performs
A[15:0]	Input A	I	Operand for the EU from the Register File (Bus A).
B[15:0]	Input B	I	Operand for the EU from the Register File (Bus B).
D [15:0]	Output D	O	Output. Connected to MUX D
zero		O	ALU Status Bit (not used in this design)

3.1.1.2 Register File

The register file contains 8, 16-bit, registers that can be written to and read from. An input called RegWrite enables write operations. The register file is connected to two output busses. Bus A is used for address output. Bus B is used for data output.

Table 4 I/O Table of the Register File.

Signal Label	**Full Name**	**I/O**	**Explanation**
regWrite	Register Write	I	High enables writing to registers
rsA[2:0]	Register, source, A	I	Register that holds operand for the EU
rsB[2:0]	Register, source, B	I	Register that holds operand for the EU
Rd [2:0]	Register, destination	I	Register in which EU op result will be stored
clk	Clock	I	Main clock signal
Bus A [15:0]		O	Connected to Bus A. Used for Jumps, RAM Wr.
Bus D [15:0]		O	Output to RAM

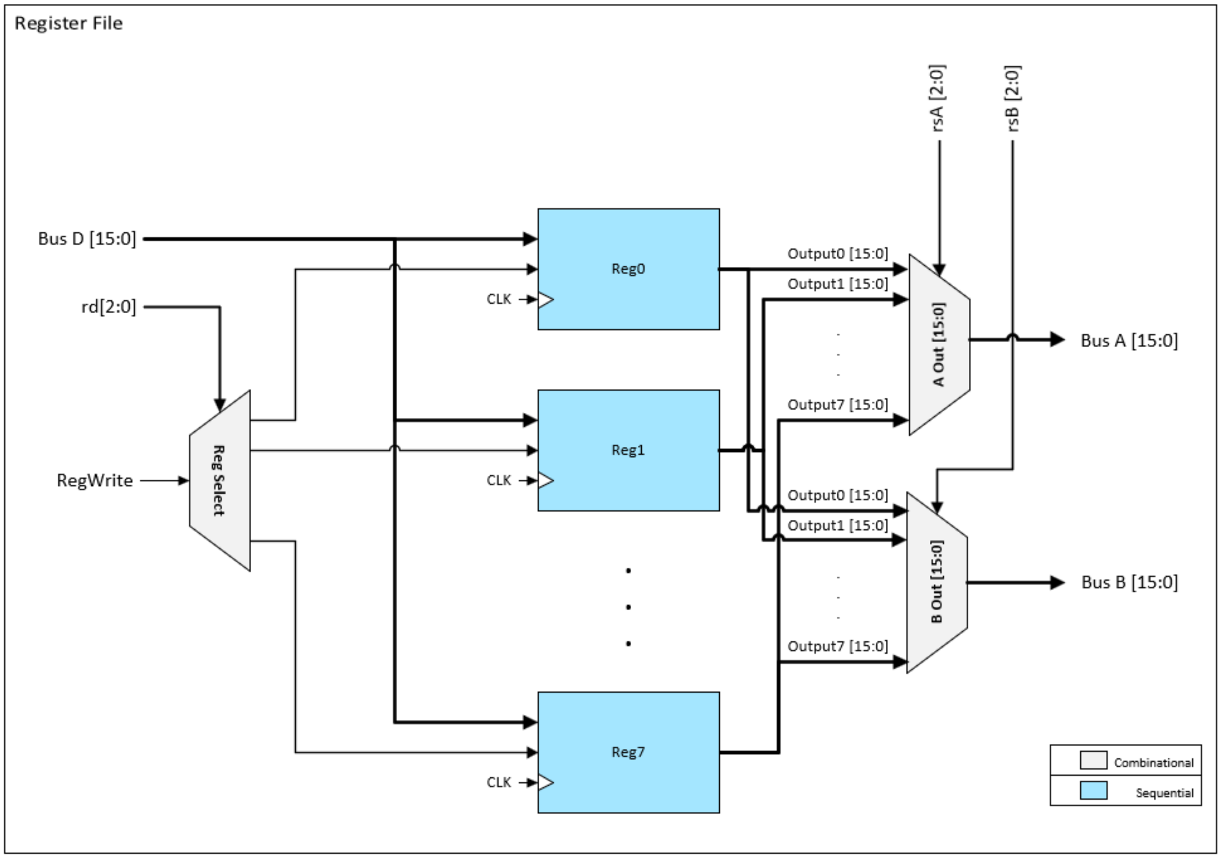


Figure 11 Architecture of the register file

3.1.2 Control Unit

The control unit contains the program counter, instruction memory, and instruction decoder. It sends signals to the datapath based on the instruction to be executed.

Table 5 I/O Table of the Control Unit

Signal Label	**Full Name**	**I/O**	**Explanation**
regWrite	Register Write	I	High enables writing to registers
rsA [2:0]	Register, source, A	I	Register that holds operand for the EU
rsB [2:0]	Register, source, B	I	Register that holds operand for the EU
Rd [2:0]	Register, destination	I	Register in which EU op result will be stored
Constant_in[2:0]		I	Used for Add Immediate instruction
MB	Multiplexer (MUX) B	I	Selects Mux B input: const_in or reg_file Out
MD	MUX D	I	Selects Mux D input: EU out or data_in
Op_select [3:0]	Operation Select	I	Selects which operation the EU performs
clk	Clock	I	Main clock signal
Address_out[15:0]		O	Connected to Bus A. Used for Jumps, RAM Wr.
Data_out[15:0]		O	Output to RAM
zero		O	ALU Status Bit (not used in this design)

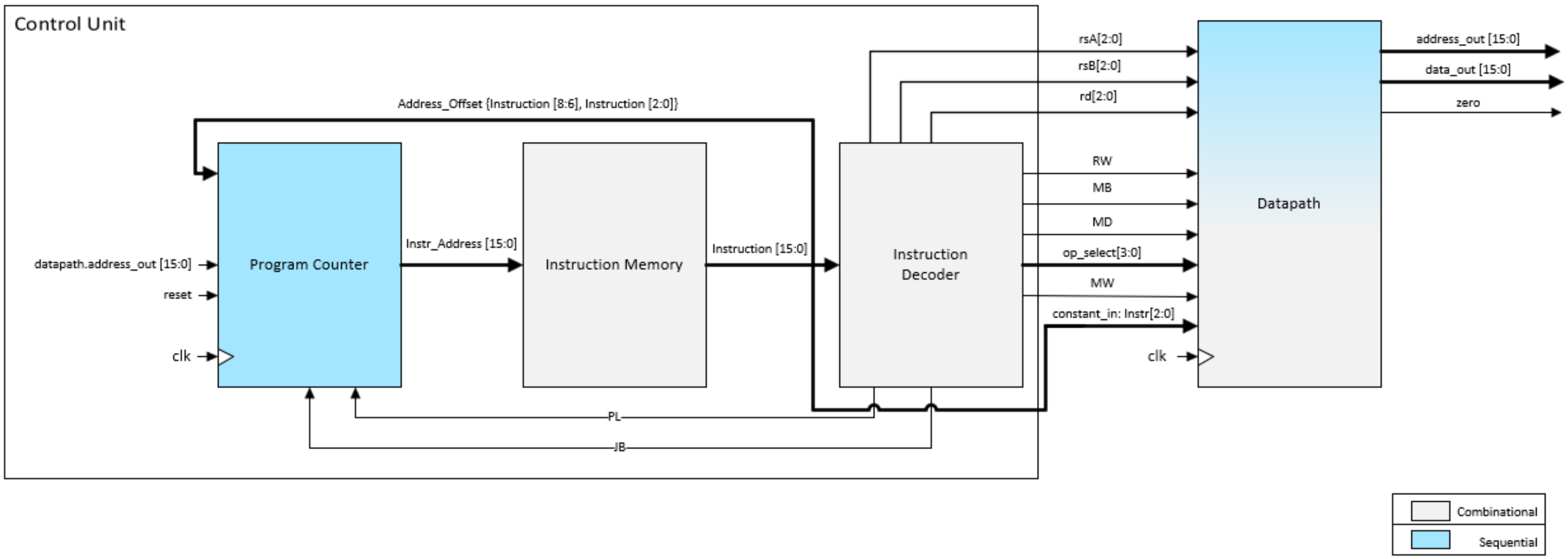


Figure 12 Architecture of the Control Unit.

3.1.2.1 Program Counter

The Program Counter provides an address to the instruction memory to determine which instruction is fetched next. It operates sequentially under normal operation but has the option to perform branch or jump instructions based on the inputs. Bits 8:6 and 2:0 of the instruction can provide an address offset for the branch instruction. The datapath's address out, or dp_address_out, is equal to bus_A.

Signal Label	**Full Name**	**I/O**	**Explanation**
address_out[15:0]		O	Connected to Bus A. Used for Jumps, RAM Wr.
Reset		I	Resets the program counter to address 0.
clk	Clock	I	Main clock signal
Address_Offset			Used for branch instructions.
PL	PC Load Enable	I	Enables offset value loading to the PC
JB	Jump Branch	I	Selects between: Jump (high) or Branch (low).
Instr_Addres [15:0]	Instruction Address	O	Instruction Address for the Instruction Memory

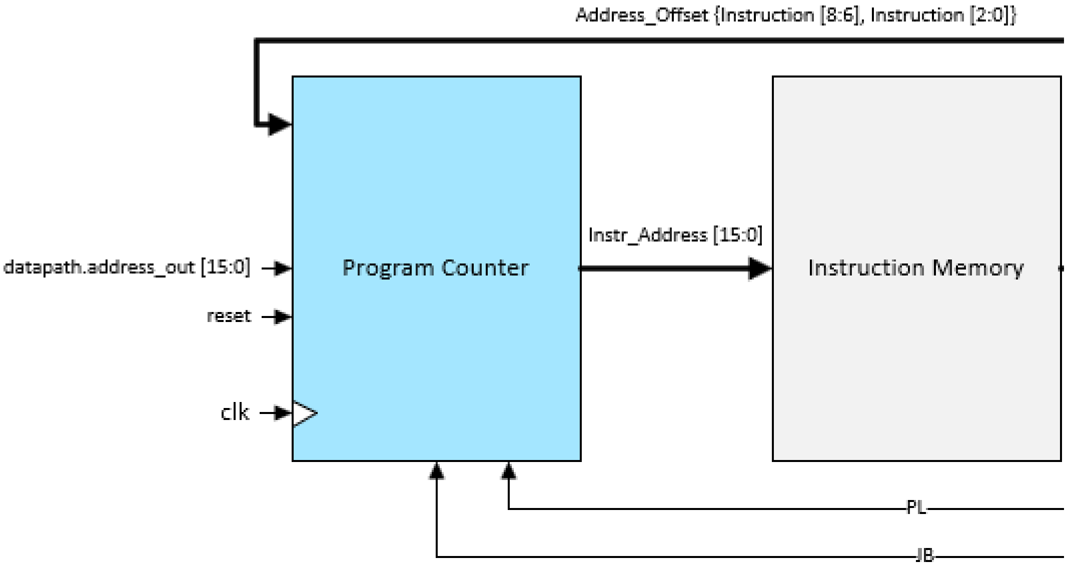

Address_Offset {Instruction [8:6], Instruction [2:0]}
datapath.address_out [15:0]
reset
clk
Program Counter
Instr_Address [15:0]
Instruction Memory
PL
JB

Figure 13 High-level overview of the Program Counter

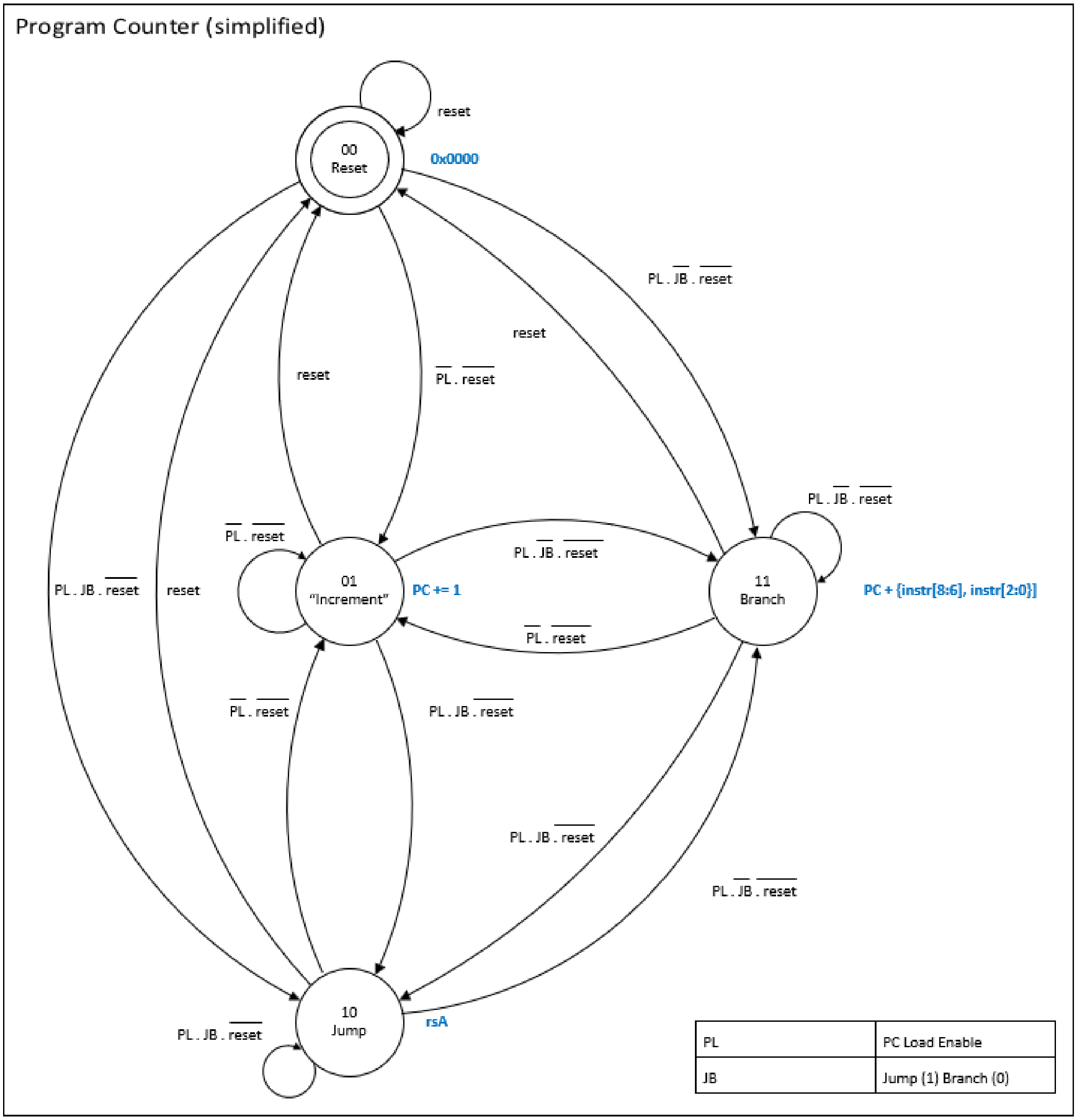

Program Counter (simplified)
reset
00
Reset
0x0000
$PL . \overline{JB} . \overline{reset}$
reset
reset
$\overline{PL} . \overline{reset}$
$PL . \overline{JB} . \overline{reset}$
$\overline{PL} . \overline{reset}$
$PL . \overline{JB} . \overline{reset}$
$PL . JB . \overline{reset}$
reset
01
"Increment"
PC += 1
11
Branch
PC + {instr[8:6], instr[2:0]}
$\overline{PL} . \overline{reset}$
$\overline{PL} . \overline{reset}$
$PL . JB . \overline{reset}$
$PL . JB . \overline{reset}$
$PL . \overline{JB} . \overline{reset}$
$PL . JB . \overline{reset}$
10
Jump
rsA
PL
PC Load Enable
JB
Jump (1) Branch (0)

Figure 14 Simplified State Diagram of the Program Counter

A simplified overview of the Program Counter's state diagram is shown in Figure 14. There are 4 main states:

- Reset: output address is all zeroes
- Increment: output address is previous address + 1.
- Branch: output address is previous address + offset value (provided in instruction bits as input to the Program Counter)
- Jump: output address is value of bus A (provided as input to the Program Counter)

Transition conditions are shown in the state diagram. A Moore implementation will be constructed for stability purposes.

Design option 1: dual states to increment, branch, and jump.
To keep incrementing, branching, or even jumping if ever required, in consecutive clock cycles, the actual FSM will have to toggle between two similar states when input signals stay constant. For example, when in the increment state, if reset and PL stay low, the FSM will transition to a second increment state and increment the output again. If reset and PL are still low the next clock cycle, the FSM will transition back to the first state and increment the output once more. A similar approach will be taken for the branch and jump states. Reset has only one state because the output remains constant (0x0000).

Design option 2: separate the FSM and address counter.
If the number of states has to be minimized, a counter with parallel load functionality could be implemented and the FSM would simply provide control signals - i.e. reset, increment, jump, or branch – for that counter. The state diagram would then match the simplified version shown above.

Taking into account implementation considerations such as the number of flops, power consumption, die size, etc. would probably lead to picking option 2. However, because back-end constraints are not in scope for this project, as described in 1.1, the first option with dual states was chosen for simplicity.

3.1.2.2 Instruction Memory

As part of the effort to keep the scope manageable, the instruction memory will be combinational to facilitate single cycle execution of fetch, decode, and execute. The instruction memory has a 16-bit address bus and contains 16-bit instructions which are formatted as described in section 2.1.2 on Instruction Formats.

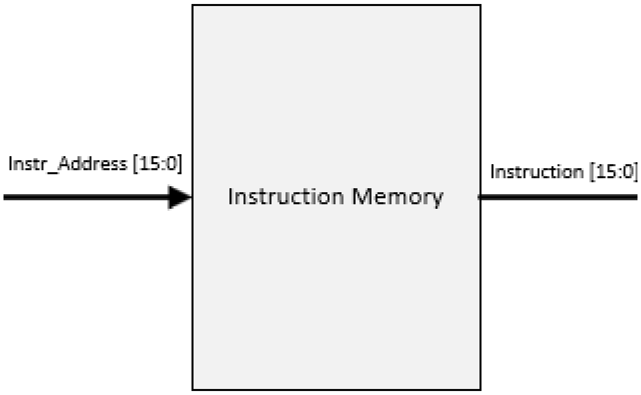


Figure 15 High-level overview of the Instruction Memory.

3.1.2.3 Instruction Decoder

As can be seen in the instruction overview in Table 2 on page 14, there are 6 major instruction types:

1. Using the Execution Unit and register(s)
2. Memory Read
3. Memory Write
4. EU with constant
5. Branch on Zero
6. Jump

Each one is referenced by 4 bits: the 3 highest MSBs and the LSB of the opcode part of the instruction: bit 9. The value of the control bits for the datapath can be deduced by comparing the instruction description in Table 2, on page 8, to the datapath overview in Figure 9 on page 17.

For example: for adding the contents of two registers, the major type is execution unit (EU) with registers. Looking at overview in Figure 9 on page 17, repeated below for reference, it can be seen that Mux B, MB, needs to be set to register input: 0. Mux D, MD, needs to be set to EU output: 0. Register Write, RW, needs to be set to 1 to allow storage of the result in the register file. Memory write, MW, is not applicable for this instruction type so the value needs to be 0. PC Load Enable, PL, needs to be zero since this is not a jump or branch. Lastly, Jump/nBranch, JB is of condition: don’t care (X).

The values for other instructions can be deduced similarly. The result for other instruction types is shown in Table 6.

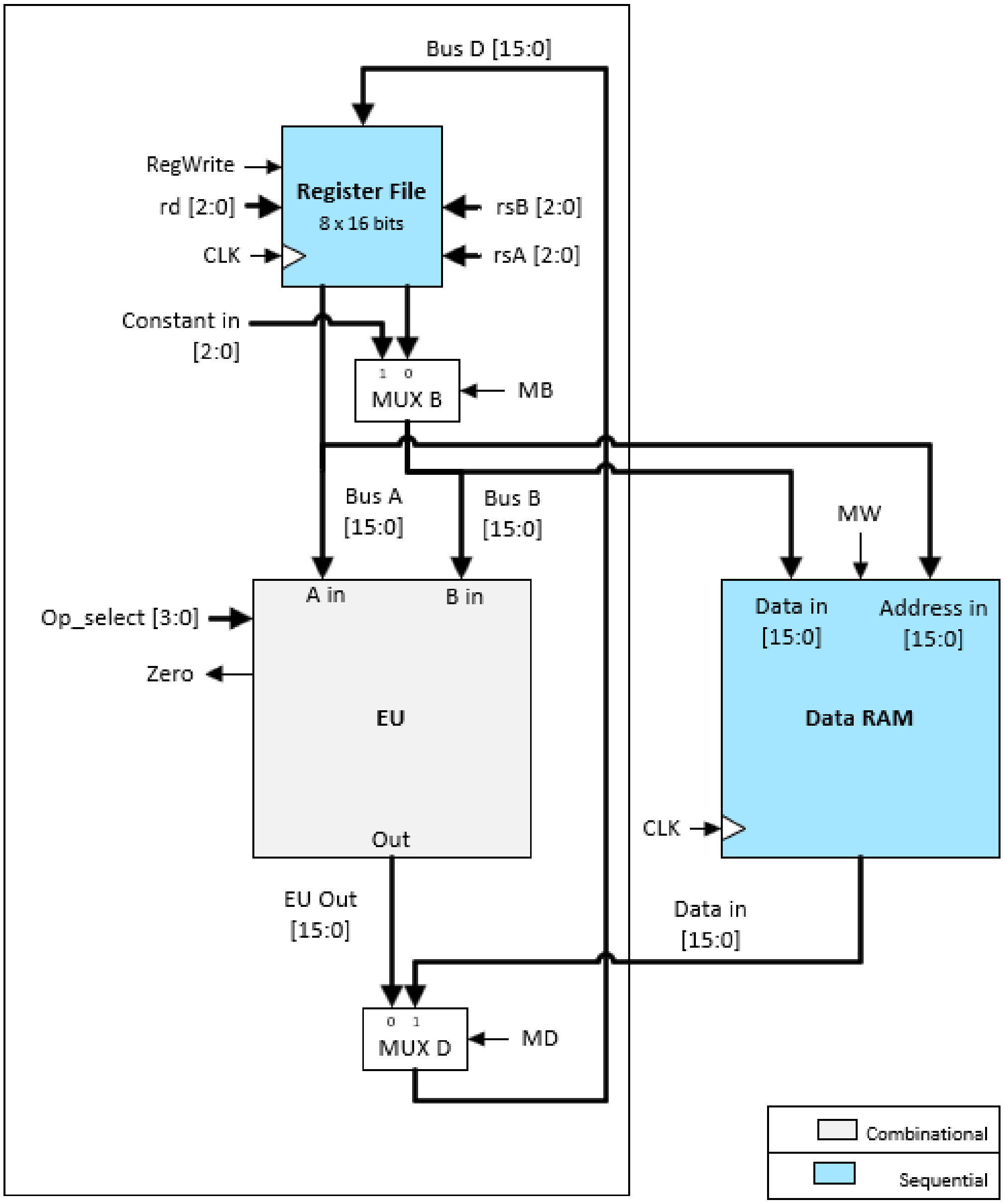


Figure 16 Datapath overview (as shown in section 3.1.1 on page 15).

Table 6 control bits for each instruction category (X: don't care)

	Instruction bits				**Control bits**						
Type	15	14	13	9	MB	MD	RW	MW	PL	JB	BC
EU with register(s)	0	0	0	X	0	0	1	0	0	X	X
Mem Read	0	0	1	X	0	1	1	0	0	X	X
Mem Write	0	1	0	X	0	X	0	1	0	X	X
EU with constant	1	0	0	X	1	0	1	0	0	X	X
Branch on Zero	1	1	0	0	X	X	0	0	1	0	0
Jump	1	1	1	X	X	X	0	0	1	1	X

Simplifying the logic of Table 6 leads to the logic for each control bit as shown below.

Table 7 Logic Formula's for Datapath Control Bits

Control Bit	Full Name	Function	Logic (instruction bits)
MB	Multiplexer (Mux) B	Mux B Selection bit	15
RW	Register Write	Enables register write	$\overline{14}$
MD	Mux D	Mux D selection bit	13
MW	Memory Write	Enable write to data RAM	$\overline{15}$ and 14
Op_Select [3:0] 3: 2: 1: 0:	Operation Select	Selects op for execution unit: 0xxx for arithmetic 10xx for logic 11xx for shifts	 12 11 10 9 and $\overline{15}$
PL	PC Load Enable	Enables address loading	15 AND 14
JB	Jump Branch	Jump (1) or Branch (0)	13
BC	Branch Condition (not used)	N/A	9
Rd[2:0]	Register, destination	Holds output of EU op	[8:6]
rsA[2:0]	Register, source, A	Holds EU operand A	[5:3]
rsB[2:0]	Register, source, B	Holds EU operand B	[2:0]

A graphical representation of the decoder is shown below. Instruction bits are shown up top and control bits, which connect to the datapath, are shown at the bottom.

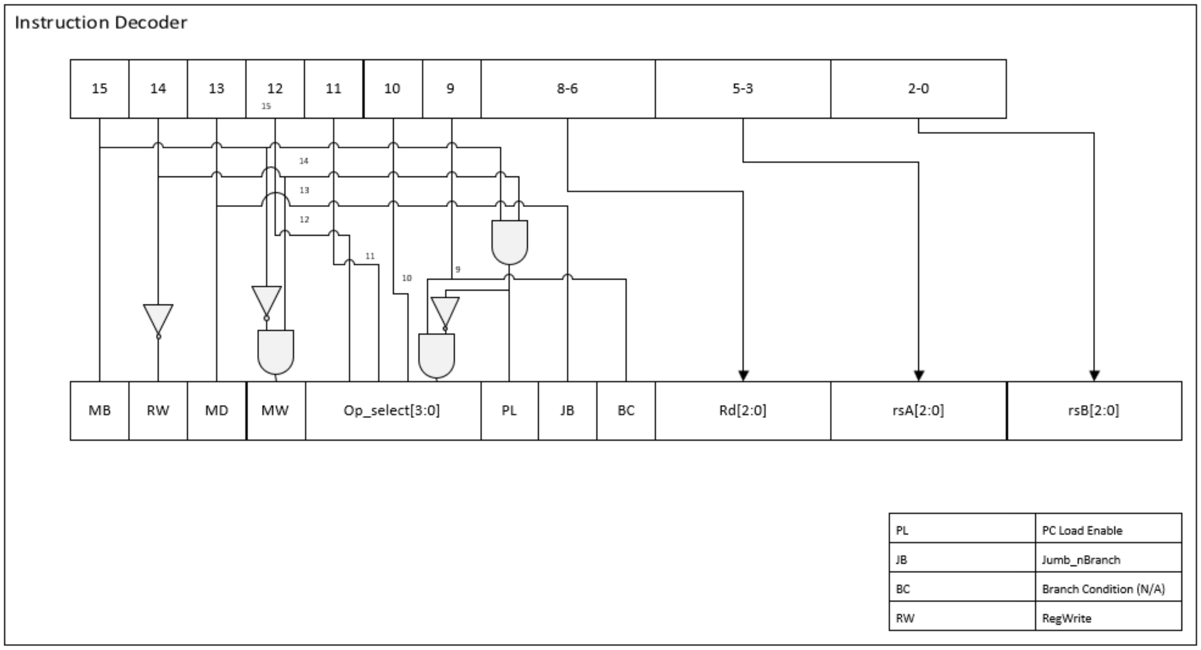


Figure 17 Instruction Decoder Logic

4 RTL Design

This chapter provides the Verilog implementation of the micro-architecture that was described in the previous chapter. Unit testbenches in Verilog that check initial functionality are described in the next chapter.

4.1 Top Level

The top level of the design holds the datapath and control unit.

4.1.1 Datapath

//The datapath's most important modules are the the Execution Unit and Register file. A graphical representation of the top layer as well as its RTL implementation are shown below.

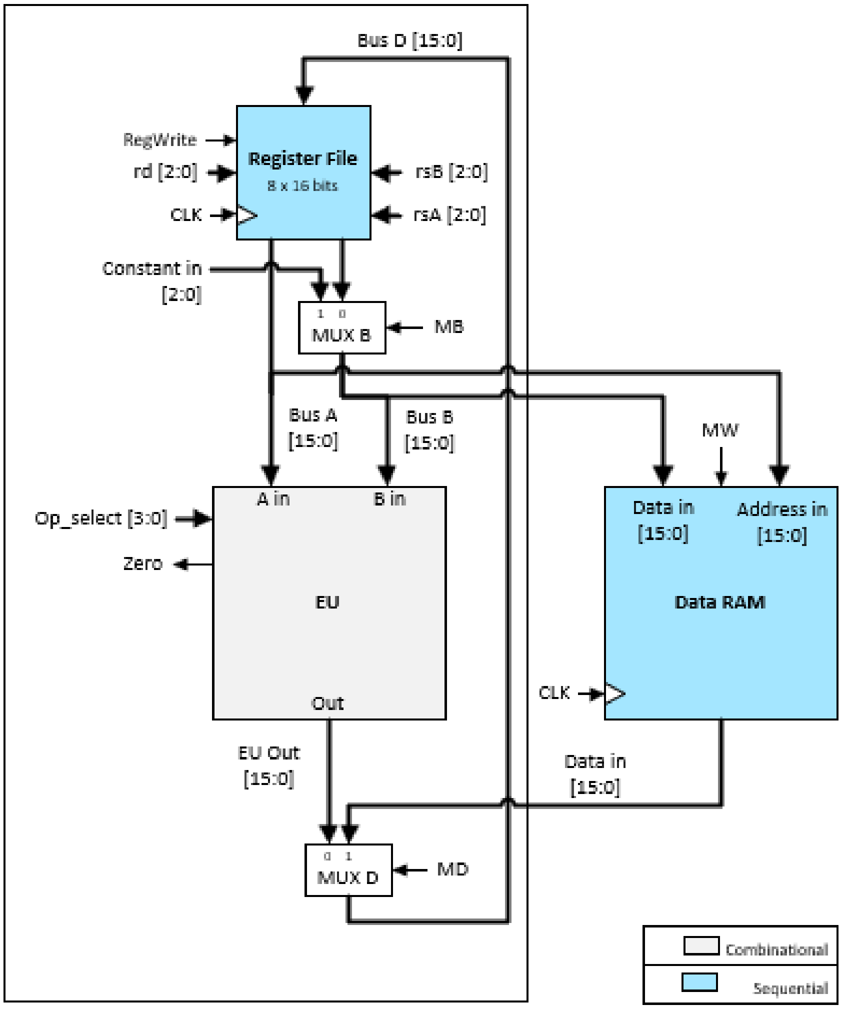
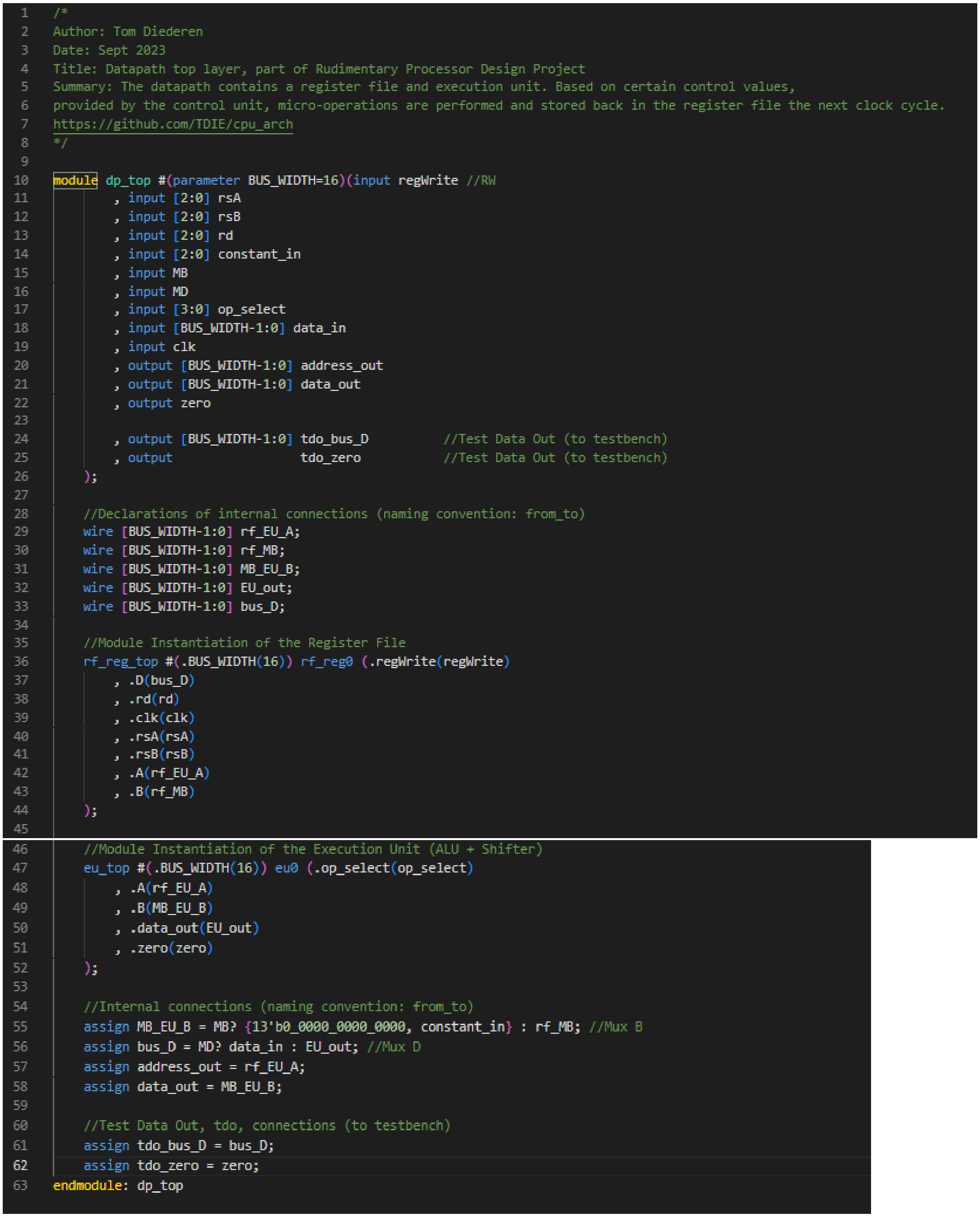


Figure 18 The datapath's top layer as shown previously in Figure 8.



```verilog
/*
Author: Tom Diederen
Date: Sept 2023
Title: Datapath top layer, part of Rudimentary Processor Design Project
Summary: The datapath contains a register file and execution unit. Based on certain control values,
provided by the control unit, micro-operations are performed and stored back in the register file the next clock cycle.
https://github.com/TDIE/cpu_arch
*/

module dp_top #(parameter BUS_WIDTH=16)(input regWrite //RW
        , input [2:0] rsA
        , input [2:0] rsB
        , input [2:0] rd
        , input [2:0] constant_in
        , input MB
        , input MD
        , input [3:0] op_select
        , input [BUS_WIDTH-1:0] data_in
        , input clk
        , output [BUS_WIDTH-1:0] address_out
        , output [BUS_WIDTH-1:0] data_out
        , output zero

        , output [BUS_WIDTH-1:0] tdo_bus_D          //Test Data Out (to testbench)
        , output                 tdo_zero           //Test Data Out (to testbench)
    );

    //Declarations of internal connections (naming convention: from_to)
    wire [BUS_WIDTH-1:0] rf_EU_A;
    wire [BUS_WIDTH-1:0] rf_MB;
    wire [BUS_WIDTH-1:0] MB_EU_B;
    wire [BUS_WIDTH-1:0] EU_out;
    wire [BUS_WIDTH-1:0] bus_D;

    //Module Instantiation of the Register File
    rf_reg_top #(.BUS_WIDTH(16)) rf_reg0 (.regWrite(regWrite)
        , .D(bus_D)
        , .rd(rd)
        , .clk(clk)
        , .rsA(rsA)
        , .rsB(rsB)
        , .A(rf_EU_A)
        , .B(rf_MB)
    );

    //Module Instantiation of the Execution Unit (ALU + Shifter)
    eu_top #(.BUS_WIDTH(16)) eu0 (.op_select(op_select)
        , .A(rf_EU_A)
        , .B(MB_EU_B)
        , .data_out(EU_out)
        , .zero(zero)
    );

    //Internal connections (naming convention: from_to)
    assign MB_EU_B = MB? {13'b0_0000_0000_0000, constant_in} : rf_MB; //Mux B
    assign bus_D = MD? data_in : EU_out; //Mux D
    assign address_out = rf_EU_A;
    assign data_out = MB_EU_B;

    //Test Data Out, tdo, connections (to testbench)
    assign tdo_bus_D = bus_D;
    assign tdo_zero = zero;
endmodule: dp_top
```

Figure 19 RTL Implementation of the Datapath's Top Layer.

4.1.1.1 Execution Unit

The RTL for the Execution unit can be divided into an Arithmetic, Logic, and Shifter section as was shown in section the section about the Execution Unit (EU)on page 18. An overview is shown again below for reference.

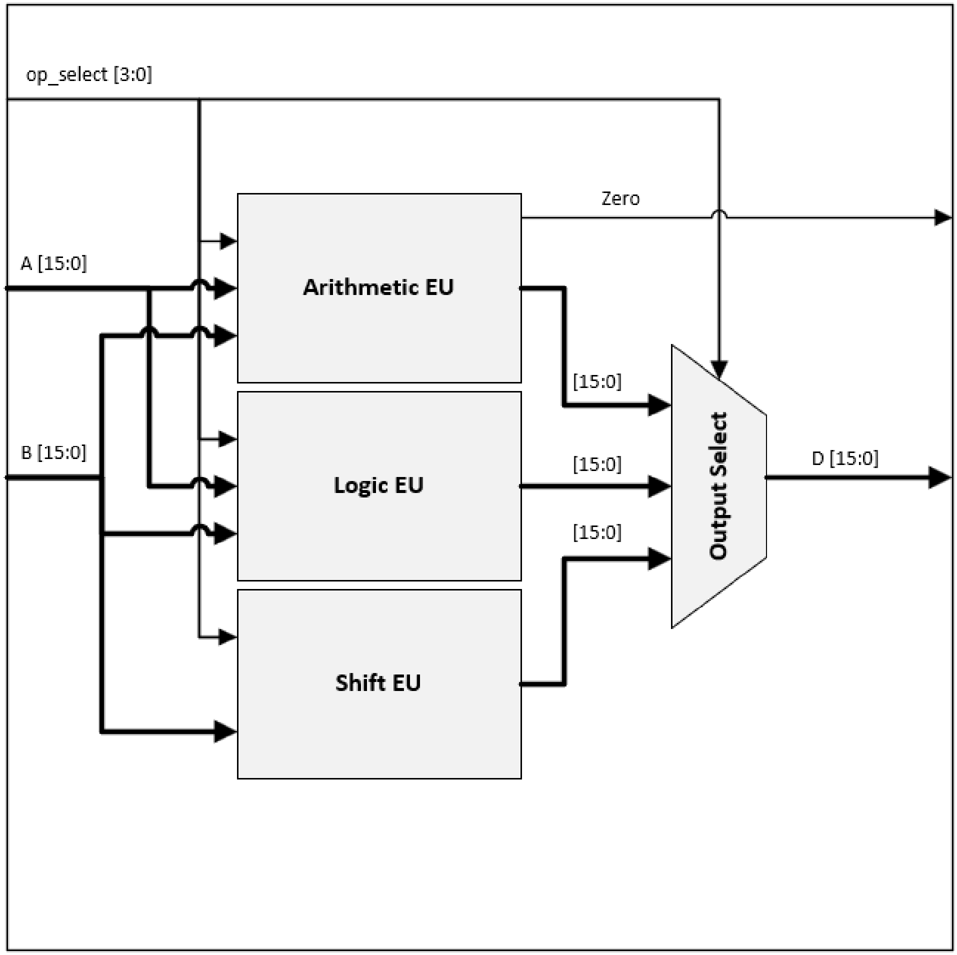
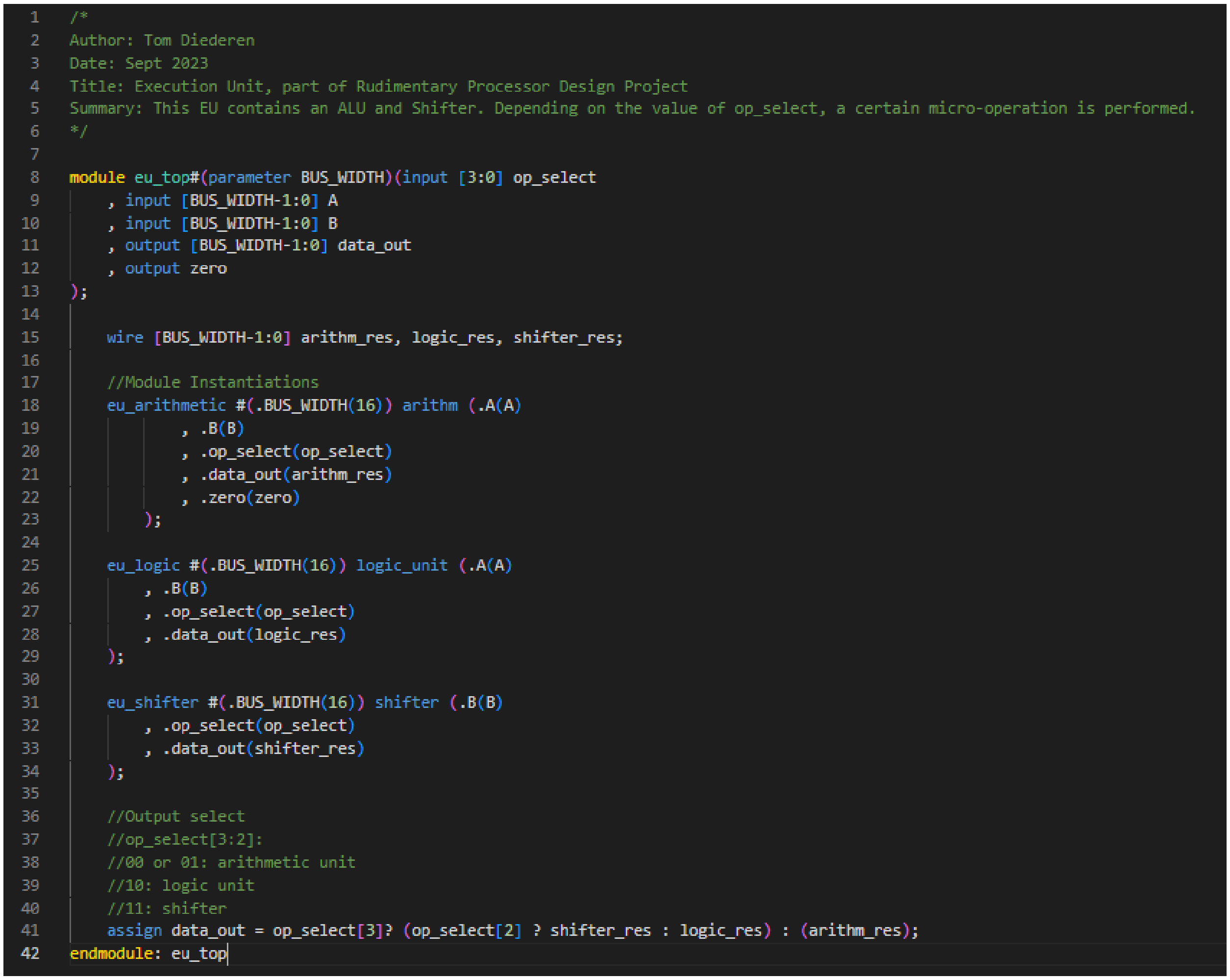


Figure 20 Overview of the architecture of the Execution Unit as shown in0 on page 18

The top level RTL is shown first, followed by the RTL of each individual block. The top level instantiates each of the 3 units and connects them together as shown above.

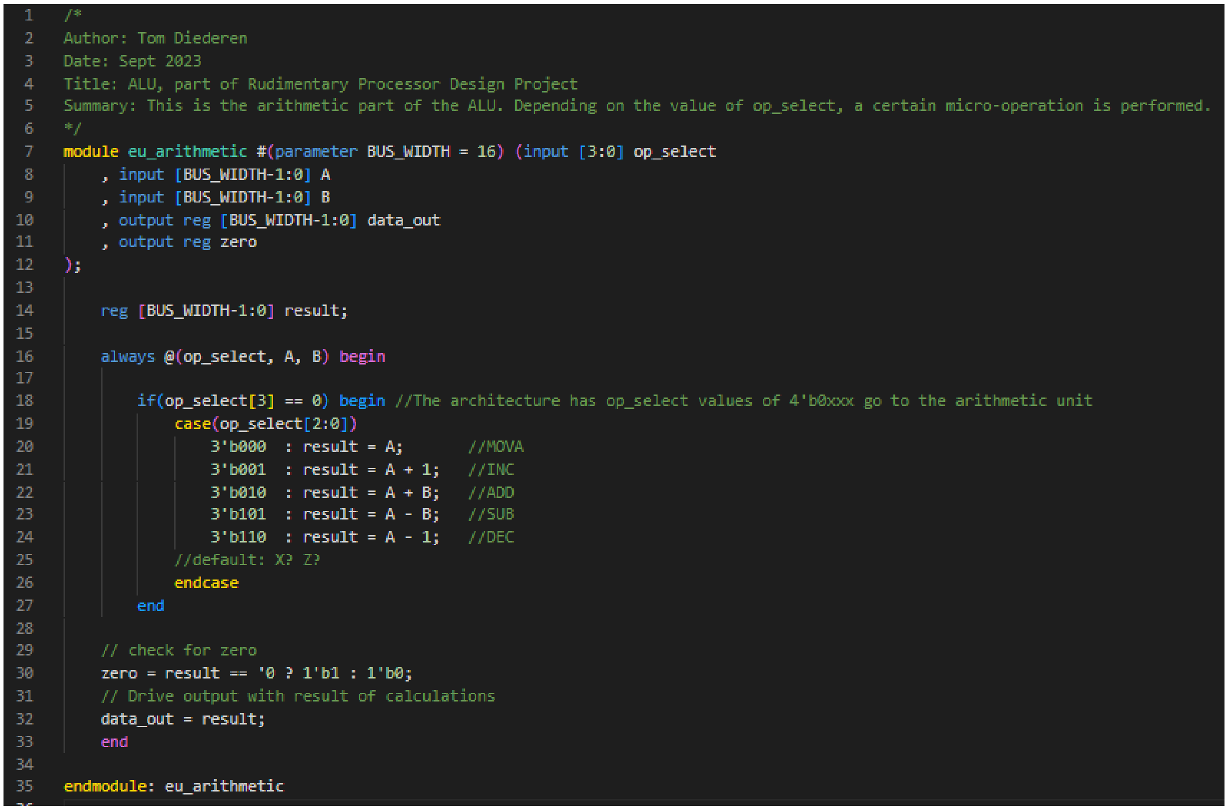


```
/*
Author: Tom Diederen
Date: Sept 2023
Title: Execution Unit, part of Rudimentary Processor Design Project
Summary: This EU contains an ALU and Shifter. Depending on the value of op_select, a certain micro-operation is performed.
*/

module eu_top#(parameter BUS_WIDTH)(input [3:0] op_select
    , input [BUS_WIDTH-1:0] A
    , input [BUS_WIDTH-1:0] B
    , output [BUS_WIDTH-1:0] data_out
    , output zero
);

    wire [BUS_WIDTH-1:0] arithm_res, logic_res, shifter_res;

    //Module Instantiations
    eu_arithmetic #(.BUS_WIDTH(16)) arithm (.A(A)
            , .B(B)
            , .op_select(op_select)
            , .data_out(arithm_res)
            , .zero(zero)
        );

    eu_logic #(.BUS_WIDTH(16)) logic_unit (.A(A)
        , .B(B)
        , .op_select(op_select)
        , .data_out(logic_res)
    );

    eu_shifter #(.BUS_WIDTH(16)) shifter (.B(B)
        , .op_select(op_select)
        , .data_out(shifter_res)
    );

    //Output select
    //op_select[3:2]:
    //00 or 01: arithmetic unit
    //10: logic unit
    //11: shifter
    assign data_out = op_select[3]? (op_select[2] ? shifter_res : logic_res) : (arithm_res);
endmodule: eu_top
```

Figure 21 Top level RTL of the Execution Unit

4.1.1.1.1 Arithmetic

The Verilog description of the arithmetic unit is shown below. It performs arithmetic operations based on the value of op_select. The design only supports unsigned operations and does not check for, or correct, overflow.

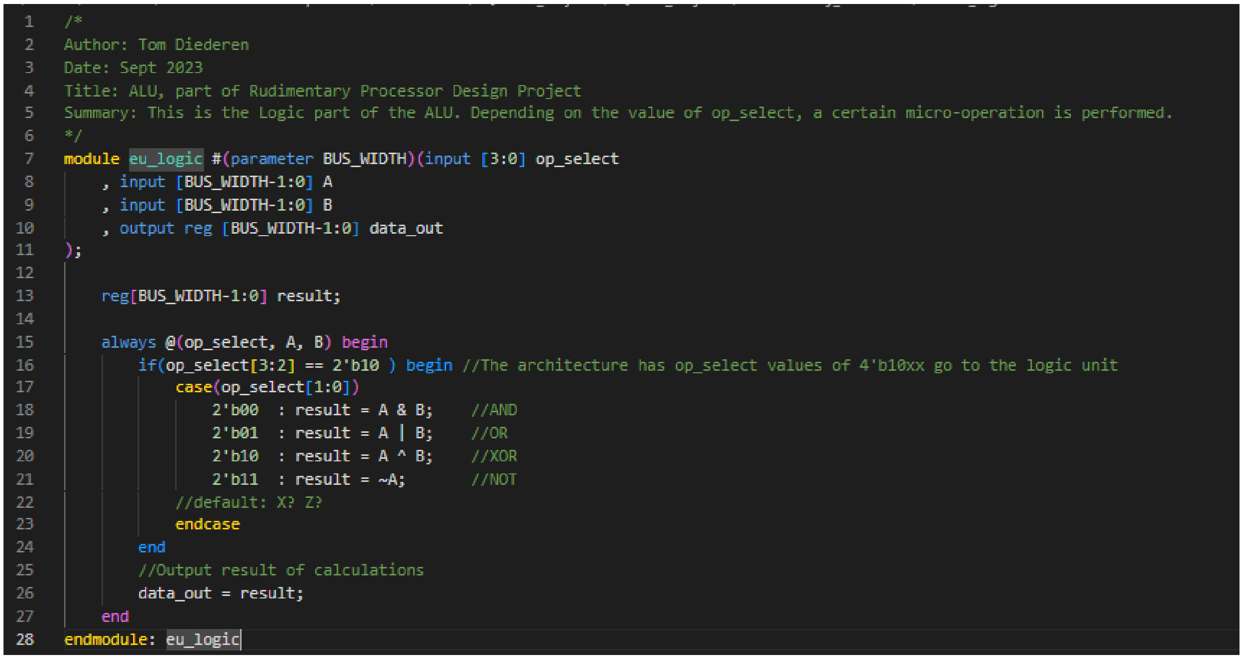


```
/*
Author: Tom Diederen
Date: Sept 2023
Title: ALU, part of Rudimentary Processor Design Project
Summary: This is the arithmetic part of the ALU. Depending on the value of op_select, a certain micro-operation is performed.
*/
module eu_arithmetic #(parameter BUS_WIDTH = 16) (input [3:0] op_select
    , input [BUS_WIDTH-1:0] A
    , input [BUS_WIDTH-1:0] B
    , output reg [BUS_WIDTH-1:0] data_out
    , output reg zero
);

    reg [BUS_WIDTH-1:0] result;

    always @(op_select, A, B) begin

        if(op_select[3] == 0) begin //The architecture has op_select values of 4'b0xxx go to the arithmetic unit
            case(op_select[2:0])
                3'b000  : result = A;       //MOVA
                3'b001  : result = A + 1;   //INC
                3'b010  : result = A + B;   //ADD
                3'b101  : result = A - B;   //SUB
                3'b110  : result = A - 1;   //DEC
            //default: X? Z?
            endcase
        end

    // check for zero
    zero = result == '0 ? 1'b1 : 1'b0;
    // Drive output with result of calculations
    data_out = result;
    end

endmodule: eu_arithmetic
```

Figure 22 RTL design of the arithmetic part of the execution unit.

4.1.1.1.2 Logic

The Verilog description of the logic unit is shown below. It performs logic operations based on the value of op_select. The instructions are shown as comments and were described in section 0 on page 14.

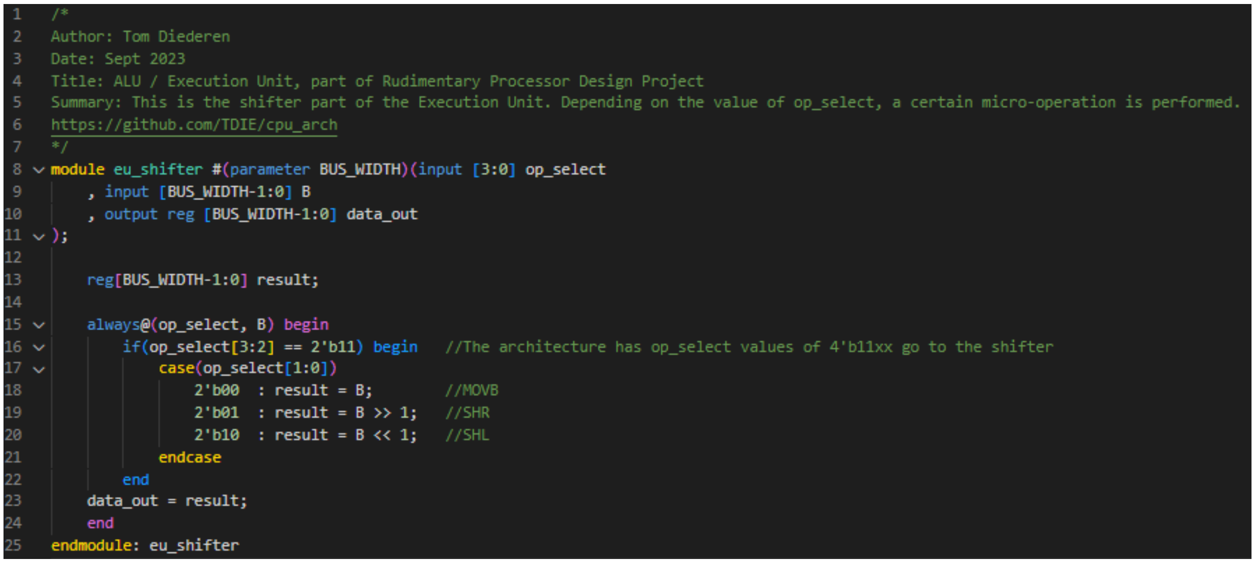


```
/*
Author: Tom Diederen
Date: Sept 2023
Title: ALU, part of Rudimentary Processor Design Project
Summary: This is the Logic part of the ALU. Depending on the value of op_select, a certain micro-operation is performed.
*/
module eu_logic #(parameter BUS_WIDTH)(input [3:0] op_select
    , input [BUS_WIDTH-1:0] A
    , input [BUS_WIDTH-1:0] B
    , output reg [BUS_WIDTH-1:0] data_out
);

    reg[BUS_WIDTH-1:0] result;

    always @(op_select, A, B) begin
        if(op_select[3:2] == 2'b10 ) begin //The architecture has op_select values of 4'b10xx go to the logic unit
            case(op_select[1:0])
                2'b00  : result = A & B;    //AND
                2'b01  : result = A | B;    //OR
                2'b10  : result = A ^ B;    //XOR
                2'b11  : result = ~A;       //NOT
            //default: X? Z?
            endcase
        end
        //Output result of calculations
        data_out = result;
    end
endmodule: eu_logic
```

Figure 23 RTL design of the logic part of the Execution Unit

4.1.1.1.3 Shifter

The Verilog description of the shifter is shown below. It performs shift operations based on the value of op_select. The instructions are referenced as comments and were described in section 0 on page 14.



```
/*
Author: Tom Diederen
Date: Sept 2023
Title: ALU / Execution Unit, part of Rudimentary Processor Design Project
Summary: This is the shifter part of the Execution Unit. Depending on the value of op_select, a certain micro-operation is performed.
https://github.com/TDIE/cpu_arch
*/
module eu_shifter #(parameter BUS_WIDTH)(input [3:0] op_select
    , input [BUS_WIDTH-1:0] B
    , output reg [BUS_WIDTH-1:0] data_out
);

    reg[BUS_WIDTH-1:0] result;

    always@(op_select, B) begin
        if(op_select[3:2] == 2'b11) begin   //The architecture has op_select values of 4'b11xx go to the shifter
            case(op_select[1:0])
                2'b00  : result = B;        //MOVB
                2'b01  : result = B >> 1;   //SHR
                2'b10  : result = B << 1;   //SHL
            endcase
        end
    data_out = result;
    end
endmodule: eu_shifter
```

Figure 24 RTL design of the shifter part of the Execution Unit

4.1.1.2 Register File

The register file contains 8, 16-bit registers.

4.1.1.2.1 Top Level

The top-level description of the register file was provided in section 0 on page 18. The diagram is shown again below for reference.

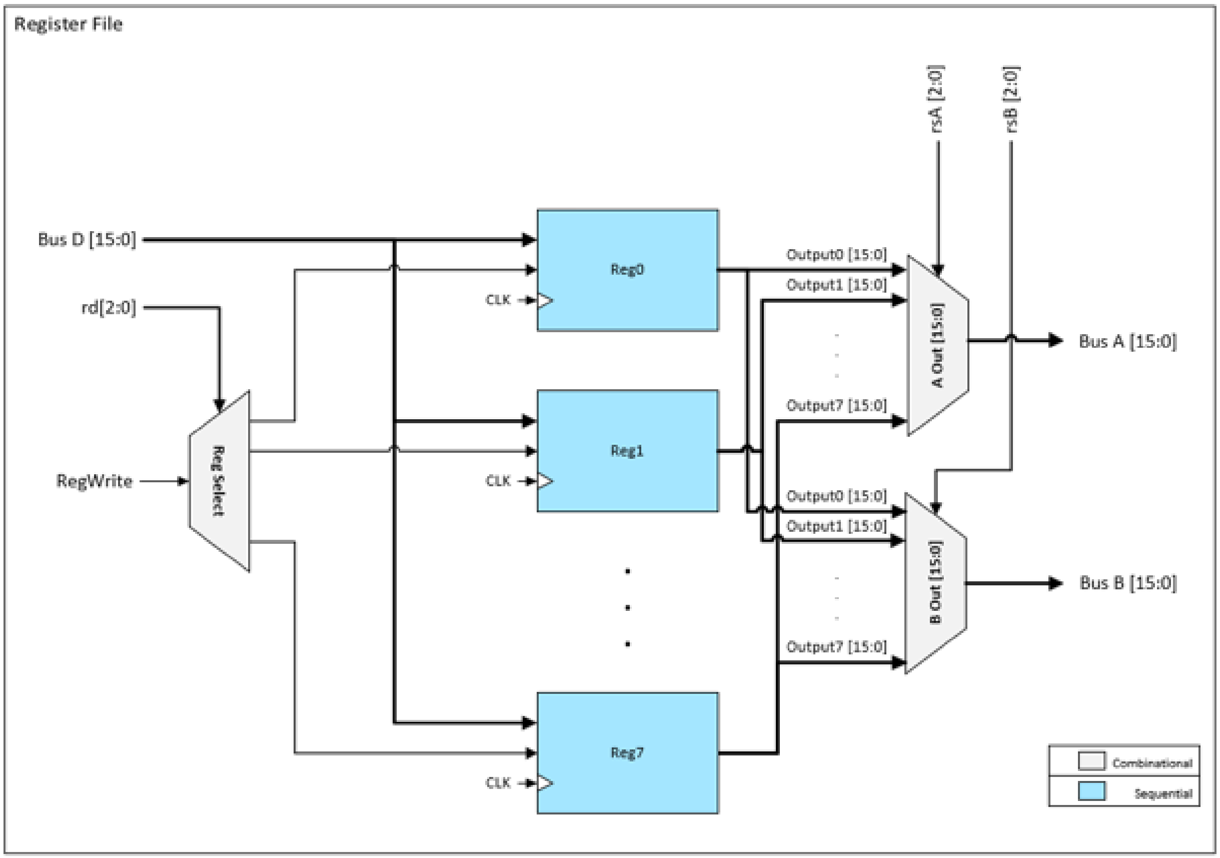
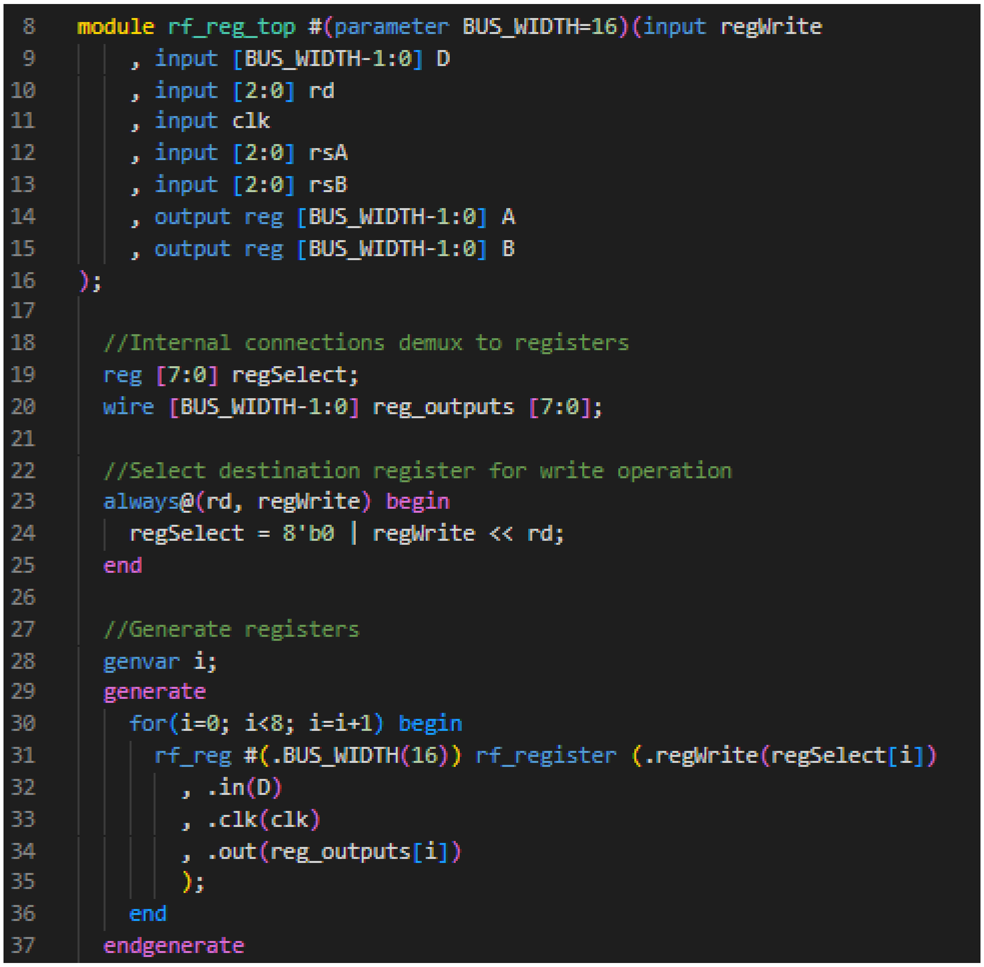


Figure 25 High level overview of the register file (as shown in section 0)

The RTL for the top level of the register file is shown below. It consists of:

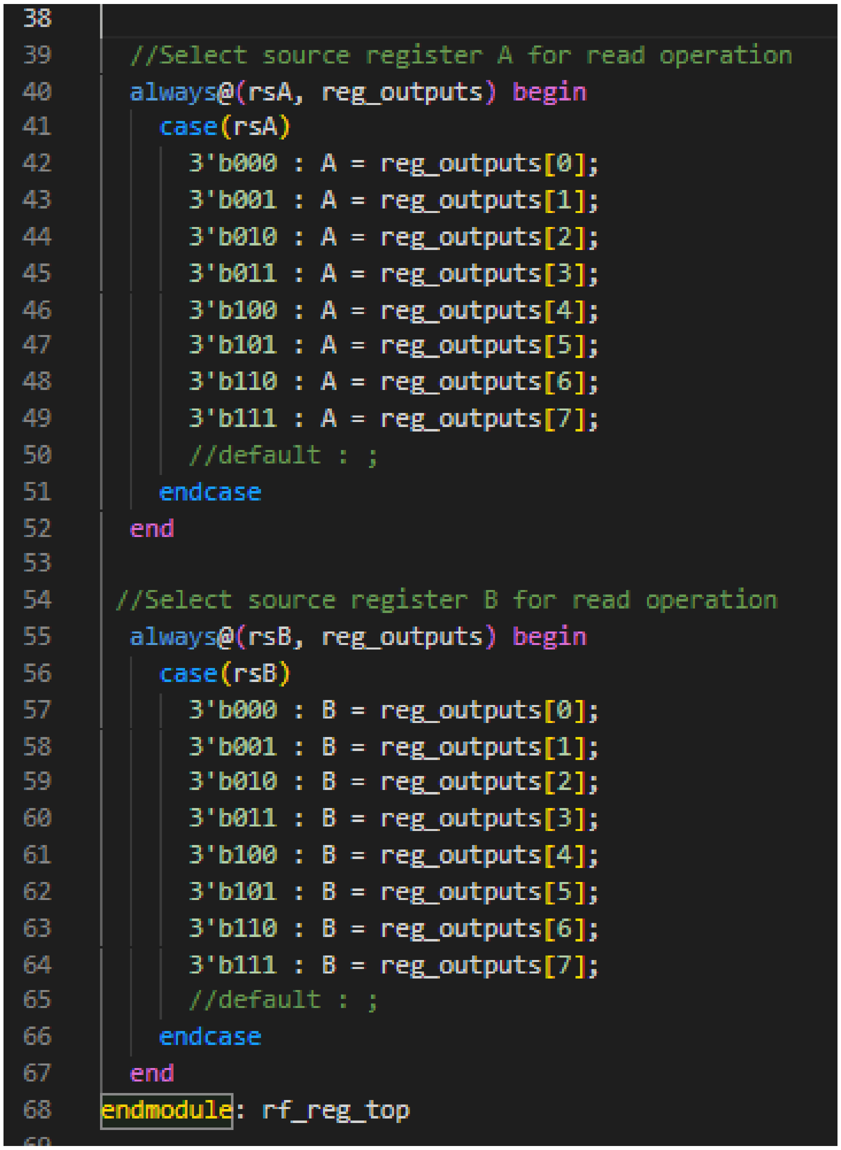
- Inputs and outputs as shown in the diagram above.
- A procedural, always, block that selects a destination register to get the status of RegWrite based on the value of the 3 bits of rd (register, destination)
- A generate block to generate 7 registers, described in the next section
- Two output procedural, always, blocks that function as multiplexers. Based on rsA, register source A, a register is selected for bus A. A similar design for bus B exists.

The Verilog description is shown below.



```verilog
module rf_reg_top #(parameter BUS_WIDTH=16)(input regWrite
    , input [BUS_WIDTH-1:0] D
    , input [2:0] rd
    , input clk
    , input [2:0] rsA
    , input [2:0] rsB
    , output reg [BUS_WIDTH-1:0] A
    , output reg [BUS_WIDTH-1:0] B
);

  //Internal connections demux to registers
  reg [7:0] regSelect;
  wire [BUS_WIDTH-1:0] reg_outputs [7:0];

  //Select destination register for write operation
  always@(rd, regWrite) begin
    regSelect = 8'b0 | regWrite << rd;
  end

  //Generate registers
  genvar i;
  generate
    for(i=0; i<8; i=i+1) begin
      rf_reg #(.BUS_WIDTH(16)) rf_register (.regWrite(regSelect[i])
        , .in(D)
        , .clk(clk)
        , .out(reg_outputs[i])
        );
    end
  endgenerate
```

Figure 26 Verilog description of the register file.

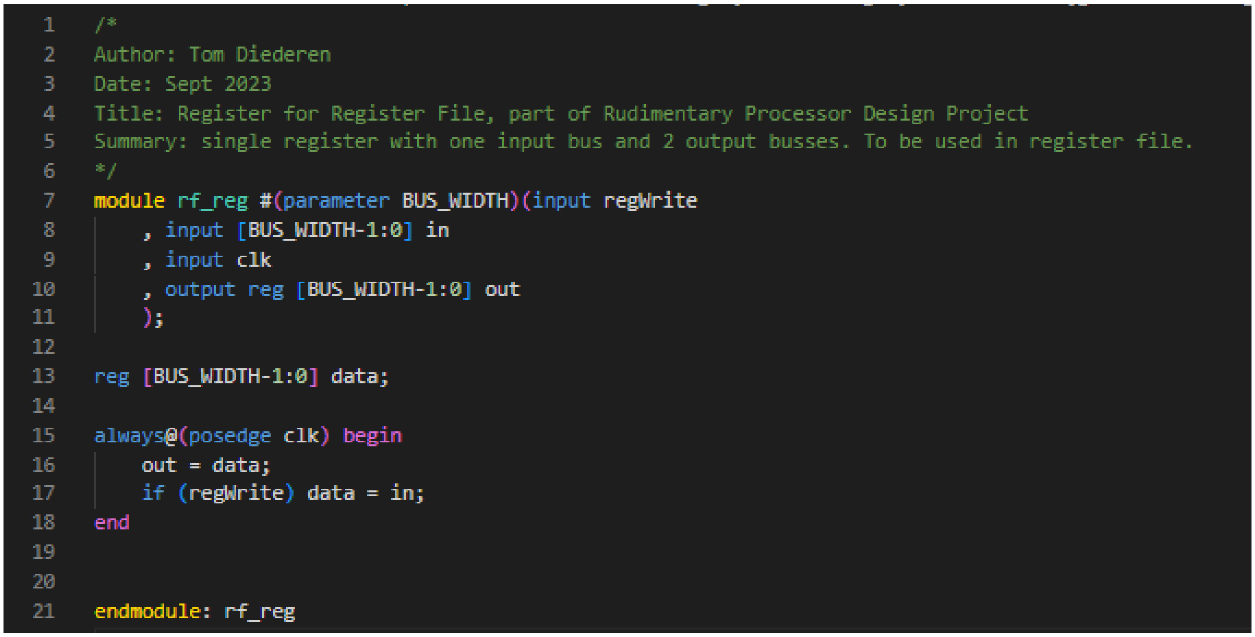


```
    //Select source register A for read operation
    always@(rsA, reg_outputs) begin
        case(rsA)
            3'b000 : A = reg_outputs[0];
            3'b001 : A = reg_outputs[1];
            3'b010 : A = reg_outputs[2];
            3'b011 : A = reg_outputs[3];
            3'b100 : A = reg_outputs[4];
            3'b101 : A = reg_outputs[5];
            3'b110 : A = reg_outputs[6];
            3'b111 : A = reg_outputs[7];
            //default : ;
        endcase
    end

   //Select source register B for read operation
    always@(rsB, reg_outputs) begin
        case(rsB)
            3'b000 : B = reg_outputs[0];
            3'b001 : B = reg_outputs[1];
            3'b010 : B = reg_outputs[2];
            3'b011 : B = reg_outputs[3];
            3'b100 : B = reg_outputs[4];
            3'b101 : B = reg_outputs[5];
            3'b110 : B = reg_outputs[6];
            3'b111 : B = reg_outputs[7];
            //default : ;
        endcase
    end
endmodule: rf_reg_top
```

Figure 27 Verilog description of the register file: output multiplexers.

4.1.1.2.2 Single Register

The RTL for an individual register is shown below.

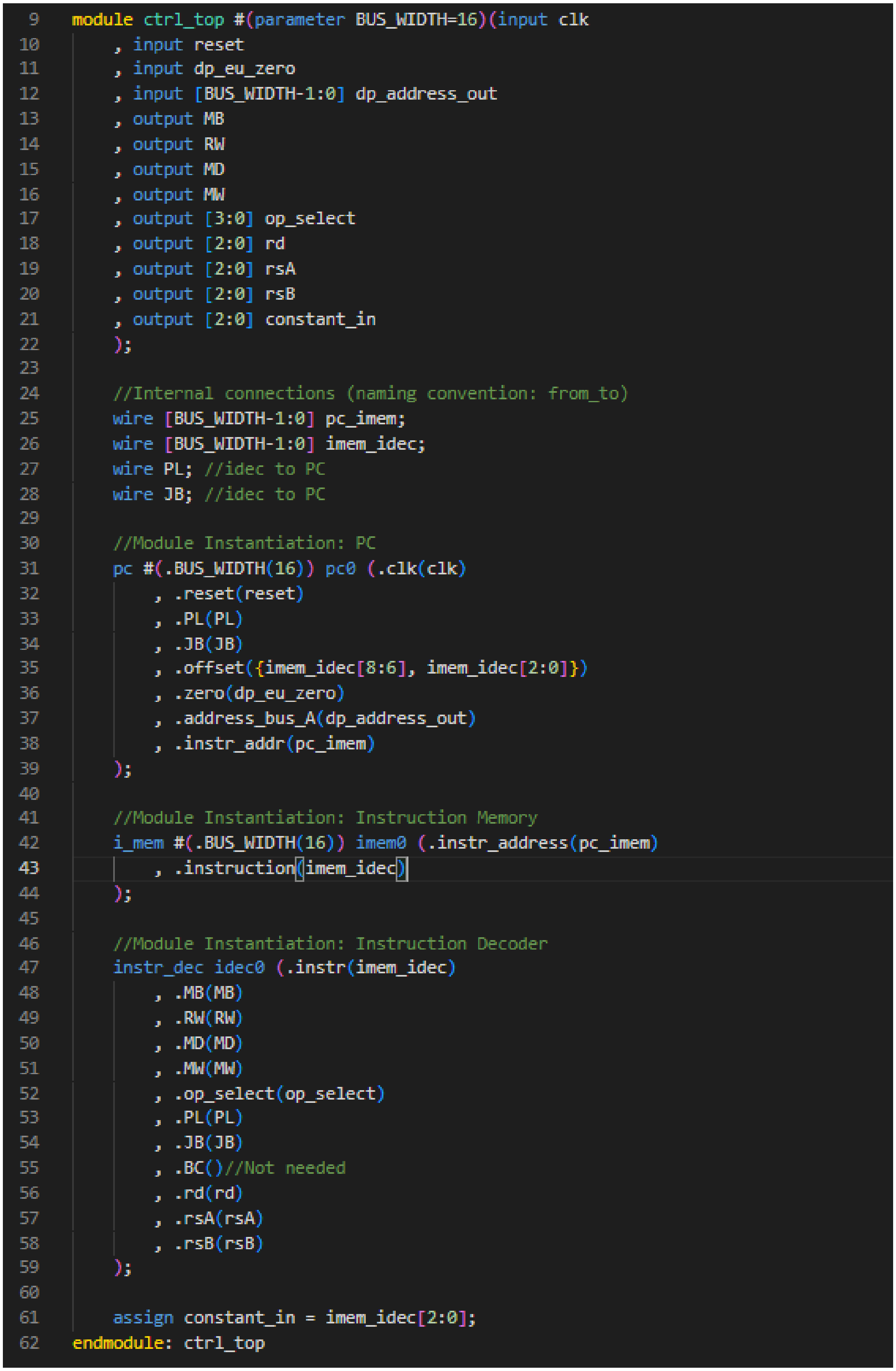


```verilog
/*
Author: Tom Diederen
Date: Sept 2023
Title: Register for Register File, part of Rudimentary Processor Design Project
Summary: single register with one input bus and 2 output busses. To be used in register file.
*/
module rf_reg #(parameter BUS_WIDTH)(input regWrite
    , input [BUS_WIDTH-1:0] in
    , input clk
    , output reg [BUS_WIDTH-1:0] out
    );

reg [BUS_WIDTH-1:0] data;

always@(posedge clk) begin
    out = data;
    if (regWrite) data = in;
end


endmodule: rf_reg
```

Figure 28 Verilog description of a single register.

4.1.2 Control Unit

The Control Unit contains the Program Counter, Instruction Memory, and Instruction Decoder. Output of the Instruction Decoder is connected to the datapath.



```verilog
module ctrl_top #(parameter BUS_WIDTH=16)(input clk
    , input reset
    , input dp_eu_zero
    , input [BUS_WIDTH-1:0] dp_address_out
    , output MB
    , output RW
    , output MD
    , output MW
    , output [3:0] op_select
    , output [2:0] rd
    , output [2:0] rsA
    , output [2:0] rsB
    , output [2:0] constant_in
    );

    //Internal connections (naming convention: from_to)
    wire [BUS_WIDTH-1:0] pc_imem;
    wire [BUS_WIDTH-1:0] imem_idec;
    wire PL; //idec to PC
    wire JB; //idec to PC

    //Module Instantiation: PC
    pc #(.BUS_WIDTH(16)) pc0 (.clk(clk)
        , .reset(reset)
        , .PL(PL)
        , .JB(JB)
        , .offset({imem_idec[8:6], imem_idec[2:0]})
        , .zero(dp_eu_zero)
        , .address_bus_A(dp_address_out)
        , .instr_addr(pc_imem)
    );

    //Module Instantiation: Instruction Memory
    i_mem #(.BUS_WIDTH(16)) imem0 (.instr_address(pc_imem)
        , .instruction(imem_idec)
    );

    //Module Instantiation: Instruction Decoder
    instr_dec idec0 (.instr(imem_idec)
        , .MB(MB)
        , .RW(RW)
        , .MD(MD)
        , .MW(MW)
        , .op_select(op_select)
        , .PL(PL)
        , .JB(JB)
        , .BC()//Not needed
        , .rd(rd)
        , .rsA(rsA)
        , .rsB(rsB)
    );

    assign constant_in = imem_idec[2:0];
endmodule: ctrl_top
```

Figure 29 Verilog description of the Control Unit's top layer.

4.1.2.1 Program Counter

The program counter determines which instruction is fetched next. It increments under normal operation but can also perform jumps and branches. A simplified state diagram is shown below.

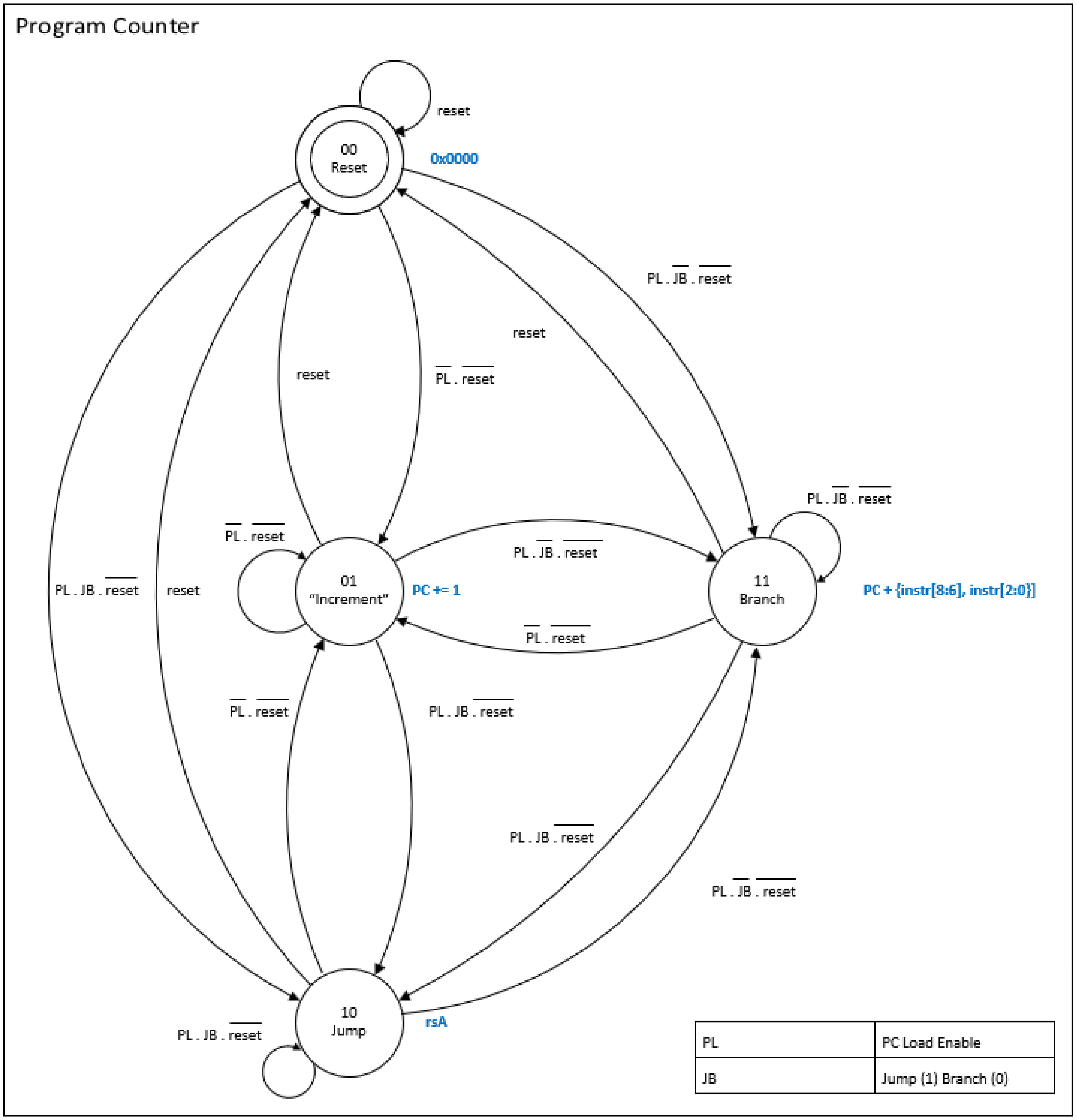
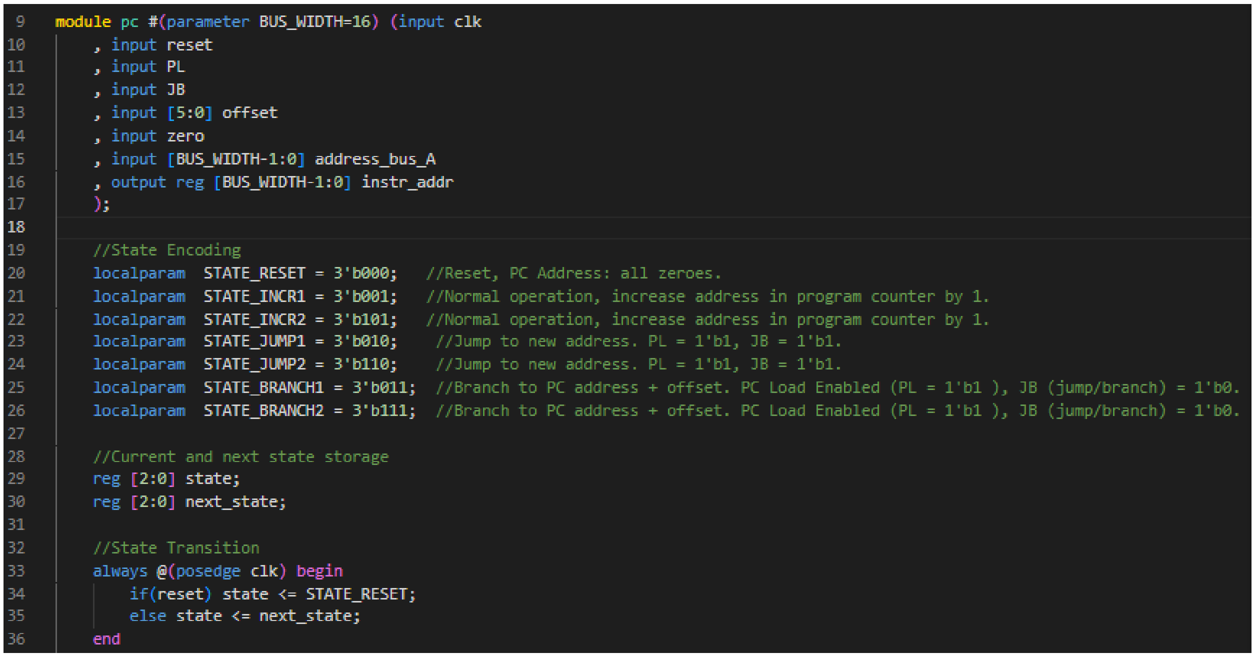
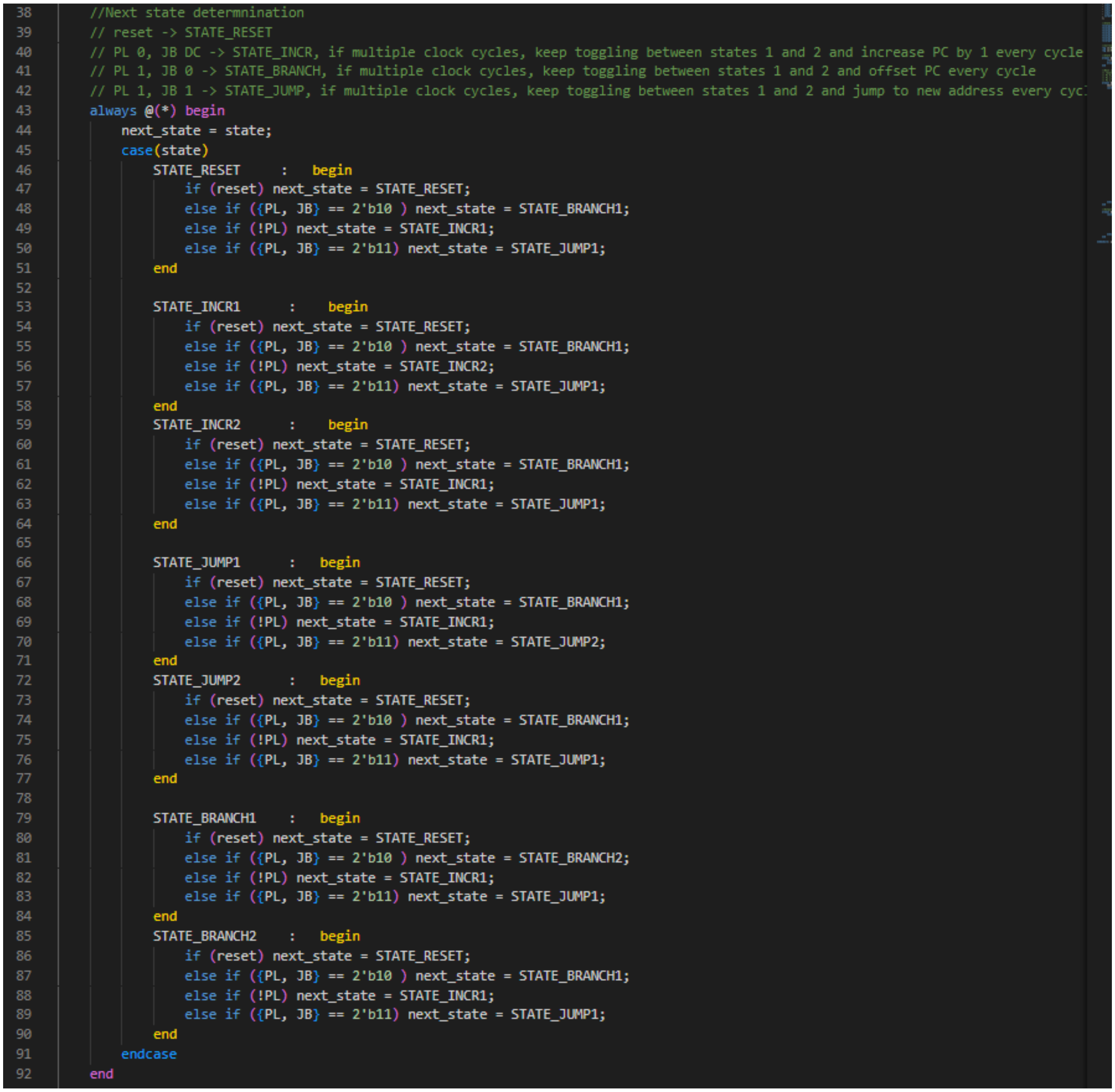

Program Counter
00
Reset
reset
0x0000
PL . $\overline{JB}$. $\overline{reset}$
reset
$\overline{PL}$. $\overline{reset}$
reset
PL . $\overline{JB}$. $\overline{reset}$
$\overline{PL}$. $\overline{reset}$
01
"Increment"
PC += 1
PL . $\overline{JB}$. $\overline{reset}$
11
Branch
PC + {instr[8:6], instr[2:0]}
$\overline{PL}$. $\overline{reset}$
PL . JB . $\overline{reset}$
reset
$\overline{PL}$. $\overline{reset}$
PL . JB . $\overline{reset}$
PL . JB . $\overline{reset}$
PL . $\overline{JB}$. $\overline{reset}$
10
Jump
rsA
PL . JB . $\overline{reset}$
PL
PC Load Enable
JB
Jump (1) Branch (0)

Figure 30 Simplified State Diagram of the Program Counter.



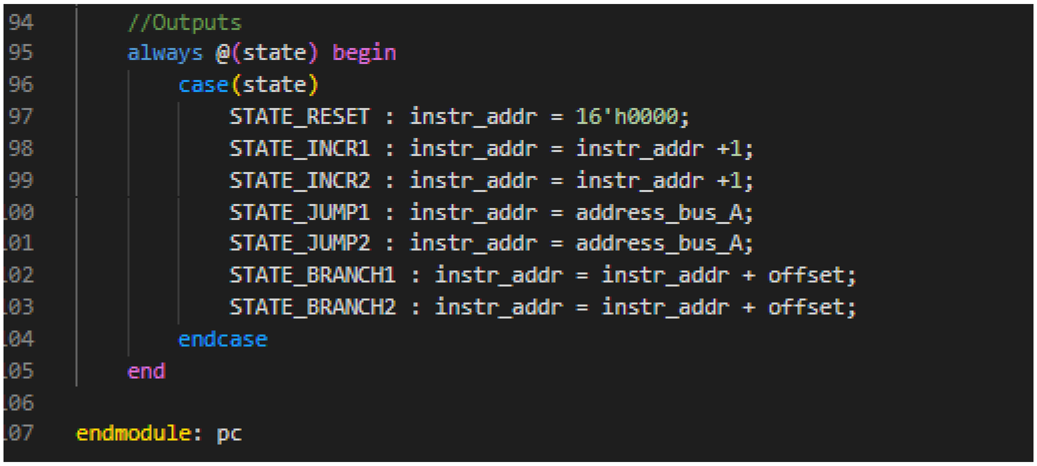
```verilog
module pc #(parameter BUS_WIDTH=16) (input clk
    , input reset
    , input PL
    , input JB
    , input [5:0] offset
    , input zero
    , input [BUS_WIDTH-1:0] address_bus_A
    , output reg [BUS_WIDTH-1:0] instr_addr
    );

    //State Encoding
    localparam  STATE_RESET = 3'b000;   //Reset, PC Address: all zeroes.
    localparam  STATE_INCR1 = 3'b001;   //Normal operation, increase address in program counter by 1.
    localparam  STATE_INCR2 = 3'b101;   //Normal operation, increase address in program counter by 1.
    localparam  STATE_JUMP1 = 3'b010;    //Jump to new address. PL = 1'b1, JB = 1'b1.
    localparam  STATE_JUMP2 = 3'b110;    //Jump to new address. PL = 1'b1, JB = 1'b1.
    localparam  STATE_BRANCH1 = 3'b011;  //Branch to PC address + offset. PC Load Enabled (PL = 1'b1 ), JB (jump/branch) = 1'b0.
    localparam  STATE_BRANCH2 = 3'b111;  //Branch to PC address + offset. PC Load Enabled (PL = 1'b1 ), JB (jump/branch) = 1'b0.

    //Current and next state storage
    reg [2:0] state;
    reg [2:0] next_state;

    //State Transition
    always @(posedge clk) begin
        if(reset) state <= STATE_RESET;
        else state <= next_state;
    end
```

Figure 31 Verilog description of the Program Counter: ports, state encoding, and state transitions.



```verilog
//Next state determnination
// reset -> STATE_RESET
// PL 0, JB DC -> STATE_INCR, if multiple clock cycles, keep toggling between states 1 and 2 and increase PC by 1 every cycle
// PL 1, JB 0 -> STATE_BRANCH, if multiple clock cycles, keep toggling between states 1 and 2 and offset PC every cycle
// PL 1, JB 1 -> STATE_JUMP, if multiple clock cycles, keep toggling between states 1 and 2 and jump to new address every cyc
always @(*) begin
    next_state = state;
    case(state)
        STATE_RESET     :   begin
            if (reset) next_state = STATE_RESET;
            else if ({PL, JB} == 2'b10 ) next_state = STATE_BRANCH1;
            else if (!PL) next_state = STATE_INCR1;
            else if ({PL, JB} == 2'b11) next_state = STATE_JUMP1;
        end

        STATE_INCR1     :    begin
            if (reset) next_state = STATE_RESET;
            else if ({PL, JB} == 2'b10 ) next_state = STATE_BRANCH1;
            else if (!PL) next_state = STATE_INCR2;
            else if ({PL, JB} == 2'b11) next_state = STATE_JUMP1;
        end
        STATE_INCR2     :    begin
            if (reset) next_state = STATE_RESET;
            else if ({PL, JB} == 2'b10 ) next_state = STATE_BRANCH1;
            else if (!PL) next_state = STATE_INCR1;
            else if ({PL, JB} == 2'b11) next_state = STATE_JUMP1;
        end

        STATE_JUMP1     :   begin
            if (reset) next_state = STATE_RESET;
            else if ({PL, JB} == 2'b10 ) next_state = STATE_BRANCH1;
            else if (!PL) next_state = STATE_INCR1;
            else if ({PL, JB} == 2'b11) next_state = STATE_JUMP2;
        end
        STATE_JUMP2     :   begin
            if (reset) next_state = STATE_RESET;
            else if ({PL, JB} == 2'b10 ) next_state = STATE_BRANCH1;
            else if (!PL) next_state = STATE_INCR1;
            else if ({PL, JB} == 2'b11) next_state = STATE_JUMP1;
        end

        STATE_BRANCH1   :   begin
            if (reset) next_state = STATE_RESET;
            else if ({PL, JB} == 2'b10 ) next_state = STATE_BRANCH2;
            else if (!PL) next_state = STATE_INCR1;
            else if ({PL, JB} == 2'b11) next_state = STATE_JUMP1;
        end
        STATE_BRANCH2   :   begin
            if (reset) next_state = STATE_RESET;
            else if ({PL, JB} == 2'b10 ) next_state = STATE_BRANCH1;
            else if (!PL) next_state = STATE_INCR1;
            else if ({PL, JB} == 2'b11) next_state = STATE_JUMP1;
        end
    endcase
end
```

Figure 32 Verilog description of the Program Counter: Next State Determination.

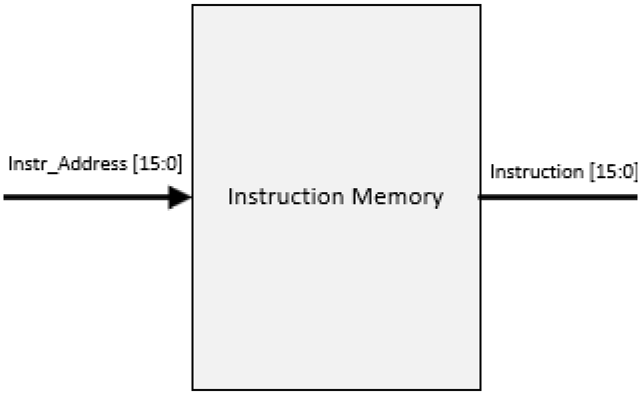


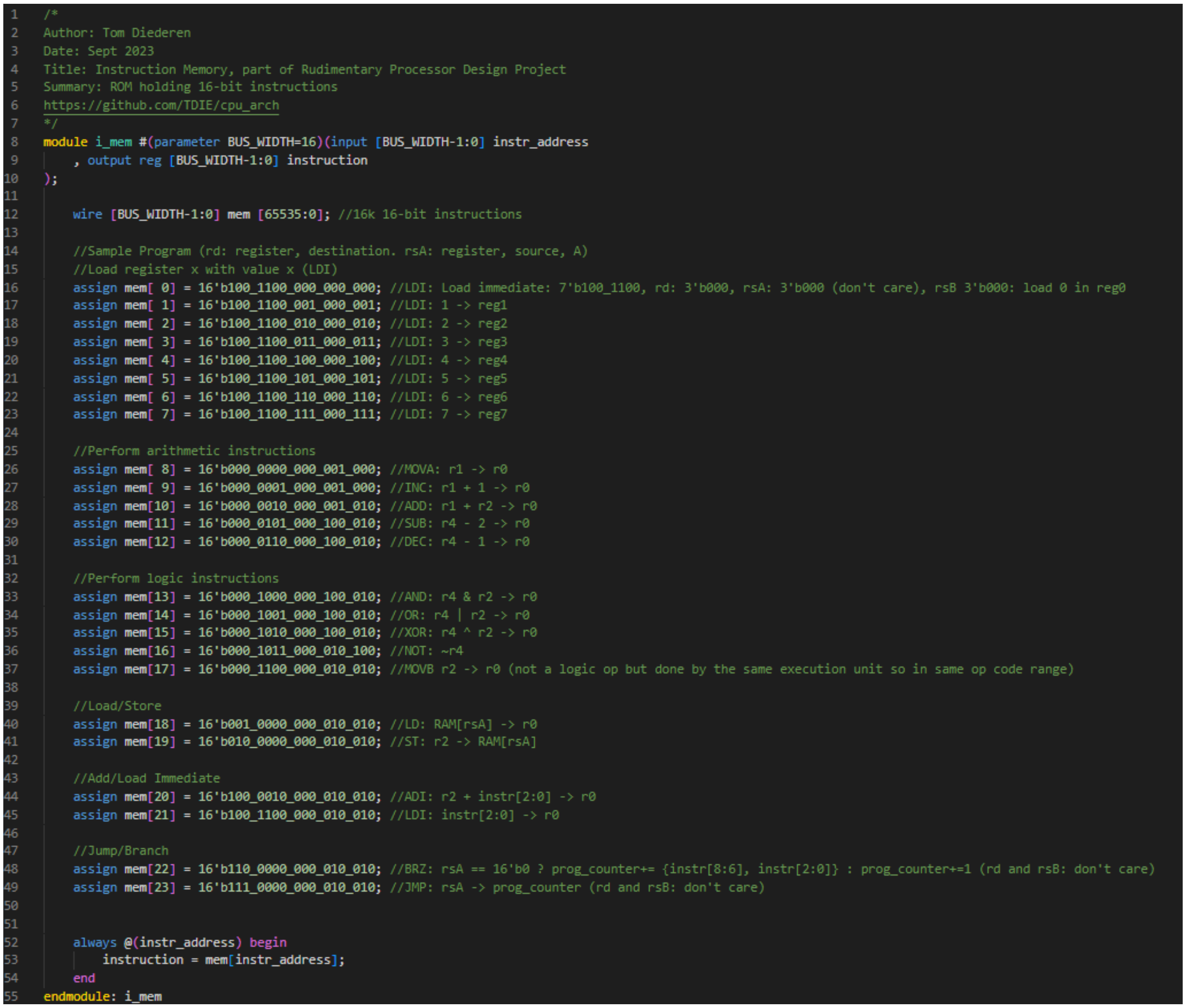
```
    //Outputs
    always @(state) begin
        case(state)
            STATE_RESET : instr_addr = 16'h0000;
            STATE_INCR1 : instr_addr = instr_addr +1;
            STATE_INCR2 : instr_addr = instr_addr +1;
            STATE_JUMP1 : instr_addr = address_bus_A;
            STATE_JUMP2 : instr_addr = address_bus_A;
            STATE_BRANCH1 : instr_addr = instr_addr + offset;
            STATE_BRANCH2 : instr_addr = instr_addr + offset;
        endcase
    end

endmodule: pc
```

Figure 33 Verilog description of the Program Counter: State Outputs.

4.1.2.2 Instruction Memory

As described in section 3.1.2.2 the instruction memory will be read-only and combinational. A sample program that performs each operation once is pre-loaded below.





```verilog
/*
Author: Tom Diederen
Date: Sept 2023
Title: Instruction Memory, part of Rudimentary Processor Design Project
Summary: ROM holding 16-bit instructions
https://github.com/TDIE/cpu_arch
*/
module i_mem #(parameter BUS_WIDTH=16)(input [BUS_WIDTH-1:0] instr_address
    , output reg [BUS_WIDTH-1:0] instruction
);

    wire [BUS_WIDTH-1:0] mem [65535:0]; //16k 16-bit instructions

    //Sample Program (rd: register, destination. rsA: register, source, A)
    //Load register x with value x (LDI)
    assign mem[ 0] = 16'b100_1100_000_000_000; //LDI: Load immediate: 7'b100_1100, rd: 3'b000, rsA: 3'b000 (don't care), rsB 3'b000: load 0 in reg0
    assign mem[ 1] = 16'b100_1100_001_000_001; //LDI: 1 -> reg1
    assign mem[ 2] = 16'b100_1100_010_000_010; //LDI: 2 -> reg2
    assign mem[ 3] = 16'b100_1100_011_000_011; //LDI: 3 -> reg3
    assign mem[ 4] = 16'b100_1100_100_000_100; //LDI: 4 -> reg4
    assign mem[ 5] = 16'b100_1100_101_000_101; //LDI: 5 -> reg5
    assign mem[ 6] = 16'b100_1100_110_000_110; //LDI: 6 -> reg6
    assign mem[ 7] = 16'b100_1100_111_000_111; //LDI: 7 -> reg7

    //Perform arithmetic instructions
    assign mem[ 8] = 16'b000_0000_000_001_000; //MOVA: r1 -> r0
    assign mem[ 9] = 16'b000_0001_000_001_000; //INC: r1 + 1 -> r0
    assign mem[10] = 16'b000_0010_000_001_010; //ADD: r1 + r2 -> r0
    assign mem[11] = 16'b000_0101_000_100_010; //SUB: r4 - 2 -> r0
    assign mem[12] = 16'b000_0110_000_100_010; //DEC: r4 - 1 -> r0

    //Perform logic instructions
    assign mem[13] = 16'b000_1000_000_100_010; //AND: r4 & r2 -> r0
    assign mem[14] = 16'b000_1001_000_100_010; //OR: r4 | r2 -> r0
    assign mem[15] = 16'b000_1010_000_100_010; //XOR: r4 ^ r2 -> r0
    assign mem[16] = 16'b000_1011_000_010_100; //NOT: ~r4
    assign mem[17] = 16'b000_1100_000_010_010; //MOVB r2 -> r0 (not a logic op but done by the same execution unit so in same op code range)

    //Load/Store
    assign mem[18] = 16'b001_0000_000_010_010; //LD: RAM[rsA] -> r0
    assign mem[19] = 16'b010_0000_000_010_010; //ST: r2 -> RAM[rsA]

    //Add/Load Immediate
    assign mem[20] = 16'b100_0010_000_010_010; //ADI: r2 + instr[2:0] -> r0
    assign mem[21] = 16'b100_1100_000_010_010; //LDI: instr[2:0] -> r0

    //Jump/Branch
    assign mem[22] = 16'b110_0000_000_010_010; //BRZ: rsA == 16'b0 ? prog_counter+= {instr[8:6], instr[2:0]} : prog_counter+=1 (rd and rsB: don't care)
    assign mem[23] = 16'b111_0000_000_010_010; //JMP: rsA -> prog_counter (rd and rsB: don't care)


    always @(instr_address) begin
        instruction = mem[instr_address];
    end
endmodule: i_mem
```

Figure 34 Verilog description of the instruction memory holding a sample program.

4.1.2.3 Instruction Decoder

As described in section 0 on page 25, the instruction decoder is combinational and provides the mapping between the instruction bits and the control bits. The Verilog implementation is shown below.

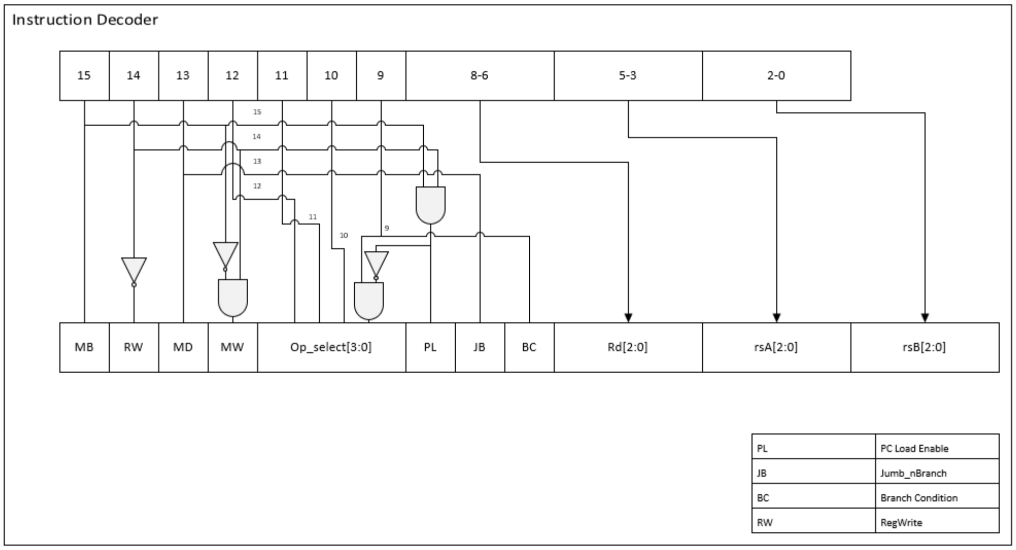
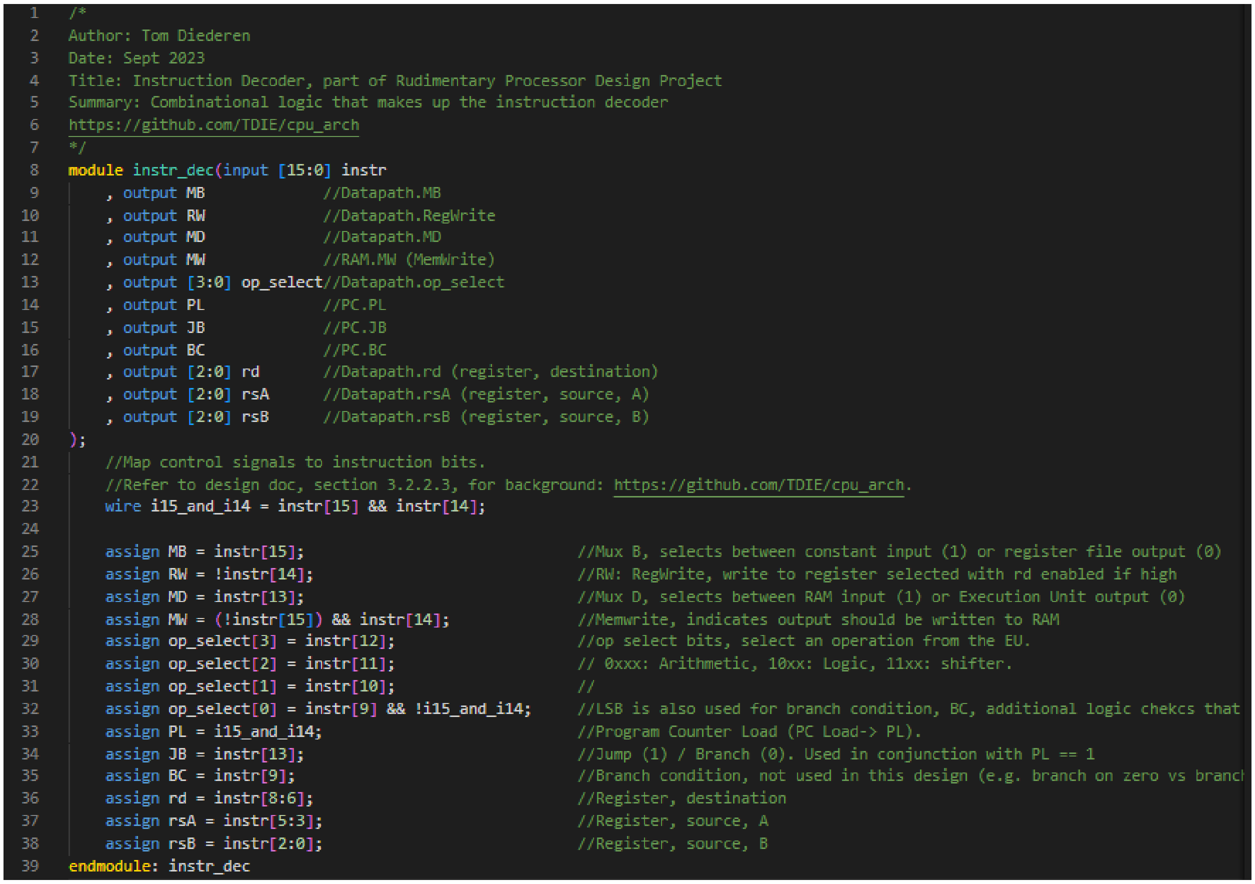


Figure 35 The Instruction Decoder as shown in 2.1.2.3 (shown here for easy reference)



```verilog
/*
Author: Tom Diederen
Date: Sept 2023
Title: Instruction Decoder, part of Rudimentary Processor Design Project
Summary: Combinational logic that makes up the instruction decoder
https://github.com/TDIE/cpu_arch
*/
module instr_dec(input [15:0] instr
    , output MB             //Datapath.MB
    , output RW             //Datapath.RegWrite
    , output MD             //Datapath.MD
    , output MW             //RAM.MW (MemWrite)
    , output [3:0] op_select//Datapath.op_select
    , output PL             //PC.PL
    , output JB             //PC.JB
    , output BC             //PC.BC
    , output [2:0] rd       //Datapath.rd (register, destination)
    , output [2:0] rsA      //Datapath.rsA (register, source, A)
    , output [2:0] rsB      //Datapath.rsB (register, source, B)
);
    //Map control signals to instruction bits.
    //Refer to design doc, section 3.2.2.3, for background: https://github.com/TDIE/cpu_arch.
    wire i15_and_i14 = instr[15] && instr[14];

    assign MB = instr[15];                              //Mux B, selects between constant input (1) or register file output (0)
    assign RW = !instr[14];                             //RW: RegWrite, write to register selected with rd enabled if high
    assign MD = instr[13];                              //Mux D, selects between RAM input (1) or Execution Unit output (0)
    assign MW = (!instr[15]) && instr[14];              //Memwrite, indicates output should be written to RAM
    assign op_select[3] = instr[12];                    //op select bits, select an operation from the EU.
    assign op_select[2] = instr[11];                    // 0xxx: Arithmetic, 10xx: Logic, 11xx: shifter.
    assign op_select[1] = instr[10];                    //
    assign op_select[0] = instr[9] && !i15_and_i14;     //LSB is also used for branch condition, BC, additional logic chekcs that
    assign PL = i15_and_i14;                            //Program Counter Load (PC Load-> PL).
    assign JB = instr[13];                              //Jump (1) / Branch (0). Used in conjunction with PL == 1
    assign BC = instr[9];                               //Branch condition, not used in this design (e.g. branch on zero vs branch
    assign rd = instr[8:6];                             //Register, destination
    assign rsA = instr[5:3];                            //Register, source, A
    assign rsB = instr[2:0];                            //Register, source, B
endmodule: instr_dec
```

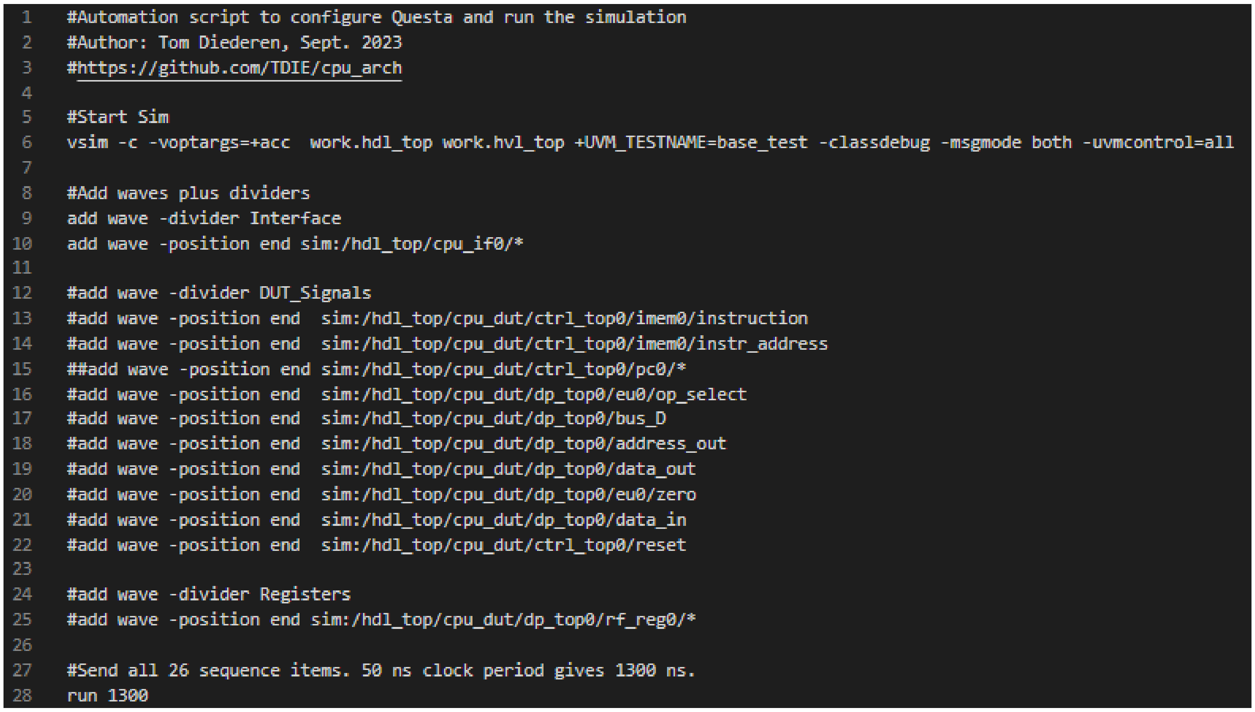
Figure 36 Verilog description of the instruction decoder.

5 VERIFICATION

This chapter describes the functional verification that was performed for this project. Unit level tests were performed by using custom written Verilog testbenches. These provide low scalability but could be implemented quickly and are sufficient for the scope of the RTL blocks of this project. System level testing through UVM is described in section 5.3 on page 73. Other verification, e.g. electrical verification or formal verification, was out of scope for this project.

5.1 EDA TOOL AUTOMATION

In order to automate repetitive commands, a simple .do file was created. One was created for each testbench to easily rerun previous tests without having to reconfigure the simulator for each test run.



```
#Automation script to configure Questa and run the simulation
#Author: Tom Diederen, Sept. 2023
#https://github.com/TDIE/cpu_arch

#Start Sim
vsim -c -voptargs=+acc  work.hdl_top work.hvl_top +UVM_TESTNAME=base_test -classdebug -msgmode both -uvmcontrol=all

#Add waves plus dividers
add wave -divider Interface
add wave -position end sim:/hdl_top/cpu_if0/*

#add wave -divider DUT_Signals
#add wave -position end  sim:/hdl_top/cpu_dut/ctrl_top0/imem0/instruction
#add wave -position end  sim:/hdl_top/cpu_dut/ctrl_top0/imem0/instr_address
##add wave -position end sim:/hdl_top/cpu_dut/ctrl_top0/pc0/*
#add wave -position end  sim:/hdl_top/cpu_dut/dp_top0/eu0/op_select
#add wave -position end  sim:/hdl_top/cpu_dut/dp_top0/bus_D
#add wave -position end  sim:/hdl_top/cpu_dut/dp_top0/address_out
#add wave -position end  sim:/hdl_top/cpu_dut/dp_top0/data_out
#add wave -position end  sim:/hdl_top/cpu_dut/dp_top0/eu0/zero
#add wave -position end  sim:/hdl_top/cpu_dut/dp_top0/data_in
#add wave -position end  sim:/hdl_top/cpu_dut/ctrl_top0/reset

#add wave -divider Registers
#add wave -position end sim:/hdl_top/cpu_dut/dp_top0/rf_reg0/*

#Send all 26 sequence items. 50 ns clock period gives 1300 ns.
run 1300
```

Figure 37 The tool automation script automatically applies settings and runs the simulation.

5.2 UNIT TESTS (VERILOG)

Each sub-block of the design, e.g. the register file, was tested by a unit testbench before being integrated into a bigger block. The results of these pre-integration tests are described in this section.

5.2.1 Datapath

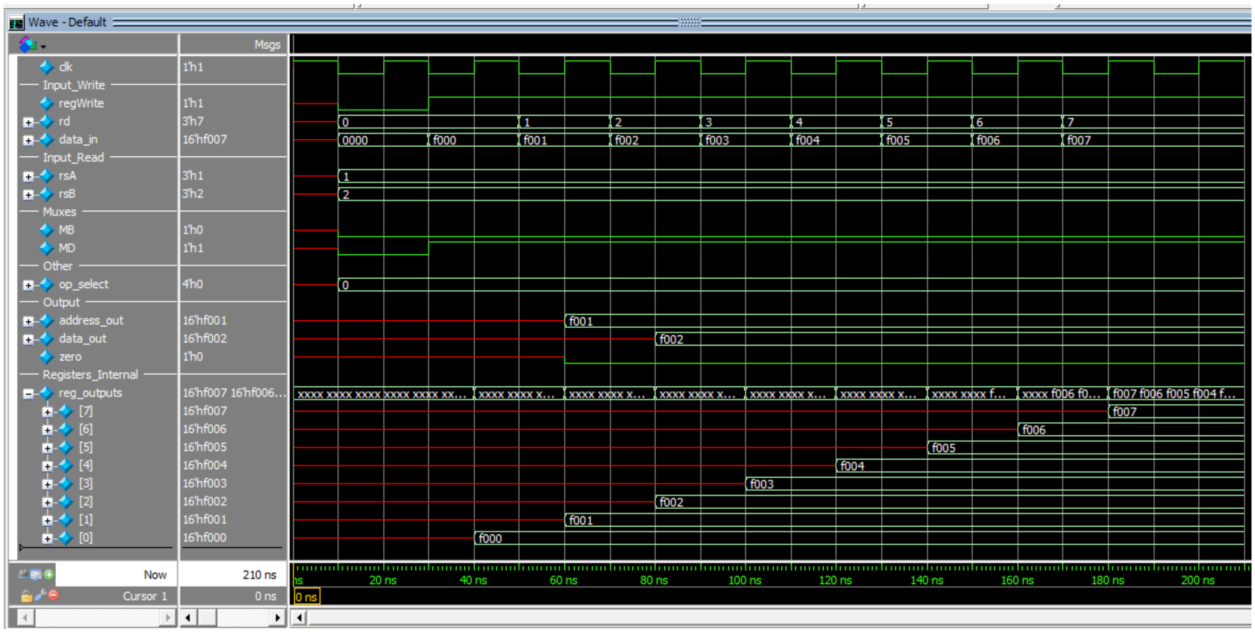
The datapath's Execution Unit and Register File were tested individually before being integrated into the datapath. The datapath itself was also tested before being integrated into the processor's overall top level.

5.2.1.1 Top Level Datapath

Test 1: Load Registers

In order to test initial functionality, all 8 registers in the register file were loaded with the value that matches their name, i.e. reg 0 contains 0, reg1 contains 1, etc. Verilog code is shown below.

Result: passed.



Wave - Default

	Msgs
clk	1'h1
— Input_Write —	
regWrite	1'h1
rd	3'h7
data_in	16'hf007
— Input_Read —	
rsA	3'h1
rsB	3'h2
— Muxes —	
MB	1'h0
MD	1'h1
— Other —	
op_select	4'h0
— Output —	
address_out	16'hf001
data_out	16'hf002
zero	1'h0
— Registers_Internal —	
reg_outputs	16'hf007 16'hf006...
[7]	16'hf007
[6]	16'hf006
[5]	16'hf005
[4]	16'hf004
[3]	16'hf003
[2]	16'hf002
[1]	16'hf001
[0]	16'hf000
Now	210 ns
Cursor 1	0 ns

Figure 38 A test of the datapath top level, loading all registers.

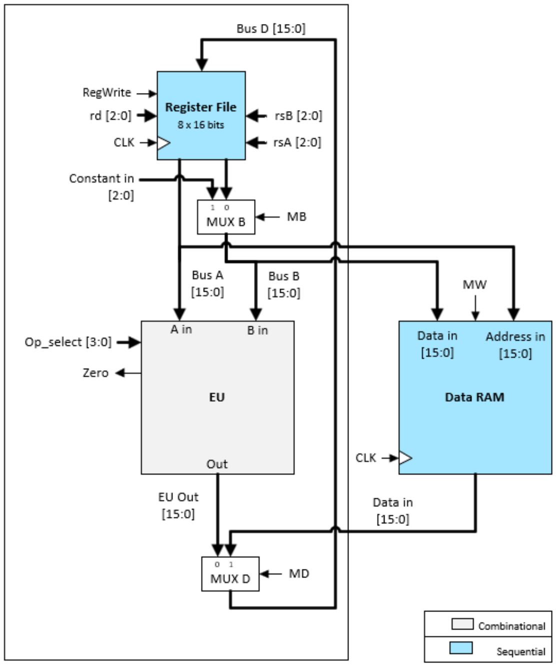
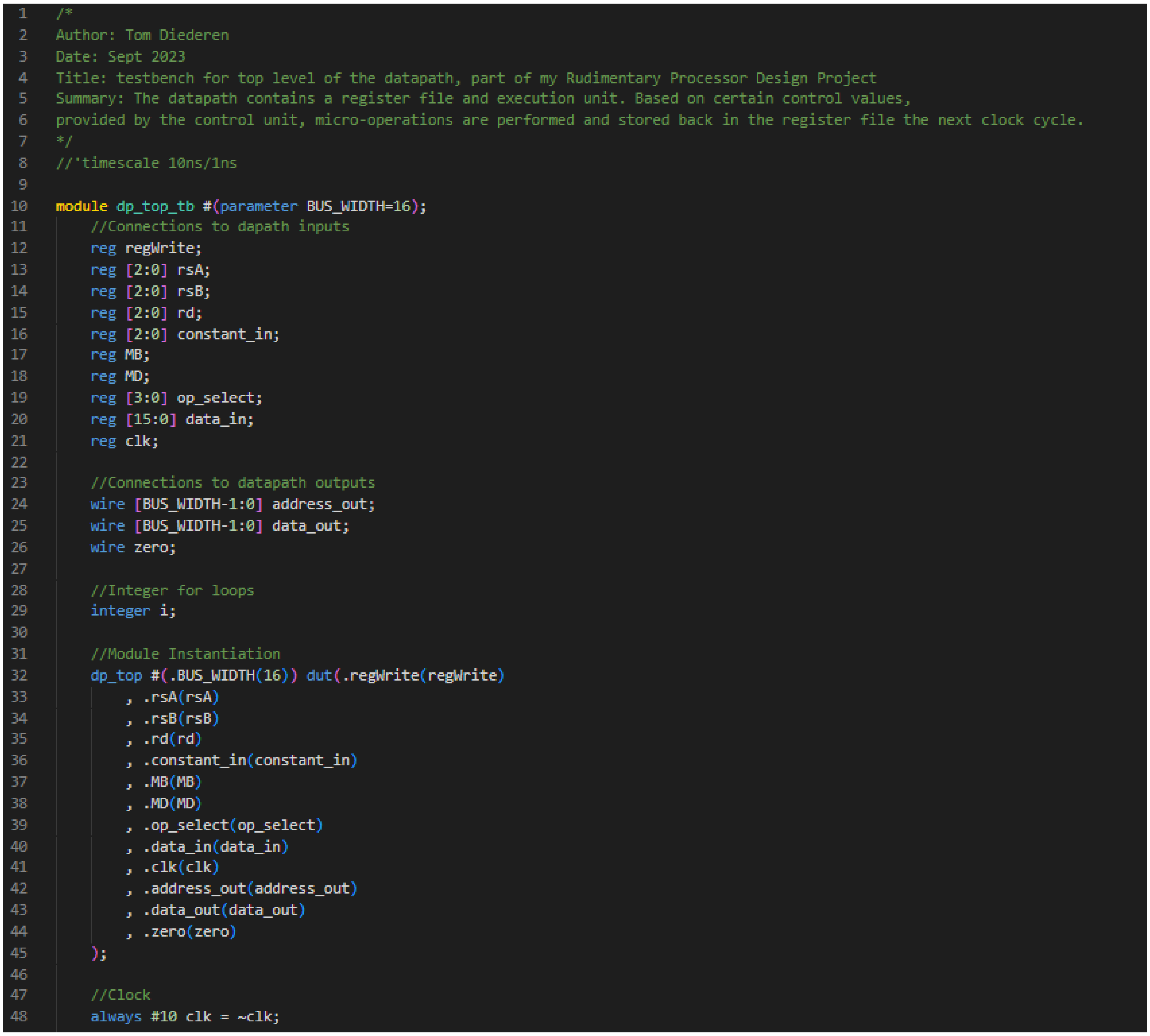
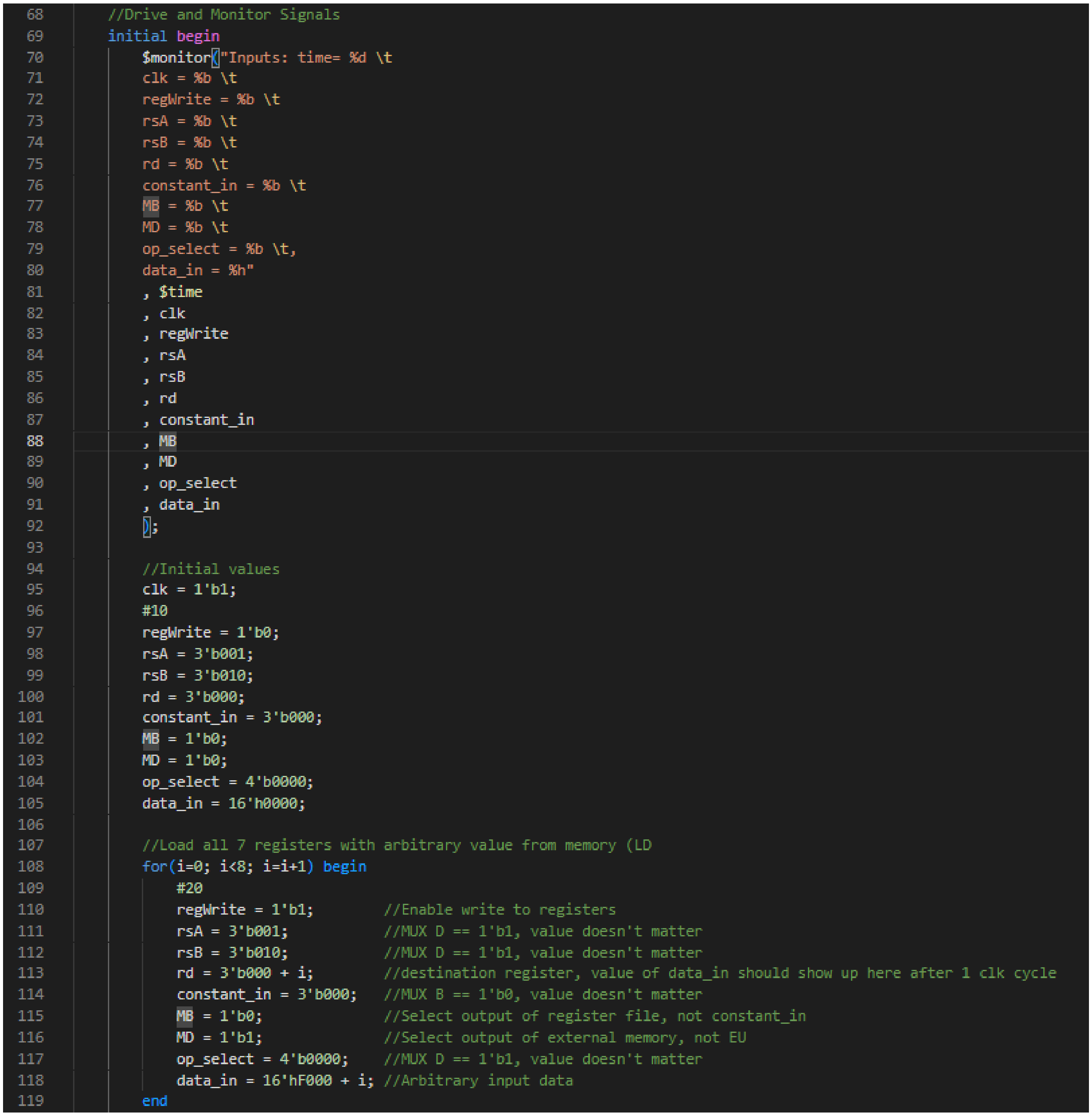


Figure 39 Top Level Overview of the Datapath



```verilog
/*
Author: Tom Diederen
Date: Sept 2023
Title: testbench for top level of the datapath, part of my Rudimentary Processor Design Project
Summary: The datapath contains a register file and execution unit. Based on certain control values,
provided by the control unit, micro-operations are performed and stored back in the register file the next clock cycle.
*/
//`timescale 10ns/1ns

module dp_top_tb #(parameter BUS_WIDTH=16);
    //Connections to dapath inputs
    reg regWrite;
    reg [2:0] rsA;
    reg [2:0] rsB;
    reg [2:0] rd;
    reg [2:0] constant_in;
    reg MB;
    reg MD;
    reg [3:0] op_select;
    reg [15:0] data_in;
    reg clk;

    //Connections to datapath outputs
    wire [BUS_WIDTH-1:0] address_out;
    wire [BUS_WIDTH-1:0] data_out;
    wire zero;

    //Integer for loops
    integer i;

    //Module Instantiation
    dp_top #(.BUS_WIDTH(16)) dut(.regWrite(regWrite)
        , .rsA(rsA)
        , .rsB(rsB)
        , .rd(rd)
        , .constant_in(constant_in)
        , .MB(MB)
        , .MD(MD)
        , .op_select(op_select)
        , .data_in(data_in)
        , .clk(clk)
        , .address_out(address_out)
        , .data_out(data_out)
        , .zero(zero)
    );

    //Clock
    always #10 clk = ~clk;
```

Figure 40 Verilog code for the datapath's top level testbench (continues below)



```verilog
    //Drive and Monitor Signals
    initial begin
        $monitor("Inputs: time= %d \t
        clk = %b \t
        regWrite = %b \t
        rsA = %b \t
        rsB = %b \t
        rd = %b \t
        constant_in = %b \t
        MB = %b \t
        MD = %b \t
        op_select = %b \t,
        data_in = %h"
        , $time
        , clk
        , regWrite
        , rsA
        , rsB
        , rd
        , constant_in
        , MB
        , MD
        , op_select
        , data_in
        );

        //Initial values
        clk = 1'b1;
        #10
        regWrite = 1'b0;
        rsA = 3'b001;
        rsB = 3'b010;
        rd = 3'b000;
        constant_in = 3'b000;
        MB = 1'b0;
        MD = 1'b0;
        op_select = 4'b0000;
        data_in = 16'h0000;

        //Load all 7 registers with arbitrary value from memory (LD
        for(i=0; i<8; i=i+1) begin
            #20
            regWrite = 1'b1;        //Enable write to registers
            rsA = 3'b001;           //MUX D == 1'b1, value doesn't matter
            rsB = 3'b010;           //MUX D == 1'b1, value doesn't matter
            rd = 3'b000 + i;        //destination register, value of data_in should show up here after 1 clk cycle
            constant_in = 3'b000;   //MUX B == 1'b0, value doesn't matter
            MB = 1'b0;              //Select output of register file, not constant_in
            MD = 1'b1;              //Select output of external memory, not EU
            op_select = 4'b0000;    //MUX D == 1'b1, value doesn't matter
            data_in = 16'hF000 + i; //Arbitrary input data
        end
```

Figure 41 Verilog code for the datapath's top level testbench (continued)

Test 2: Execute Instructions

A single test is performed for each instruction type that involves the datapath (so no memory writes and jump or branch instructions). The Verilog code is shown below.

Result: passed.

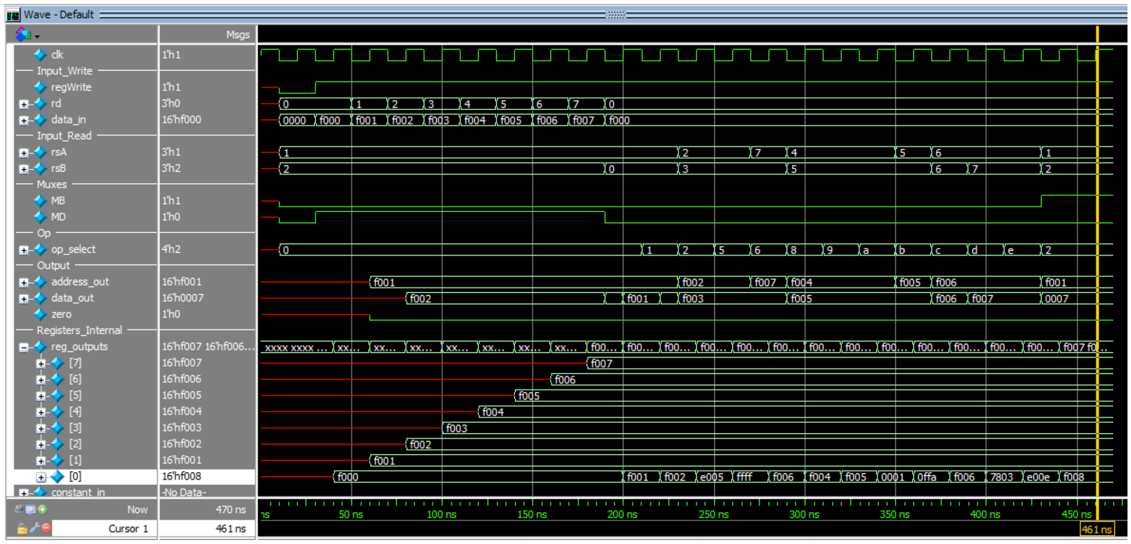
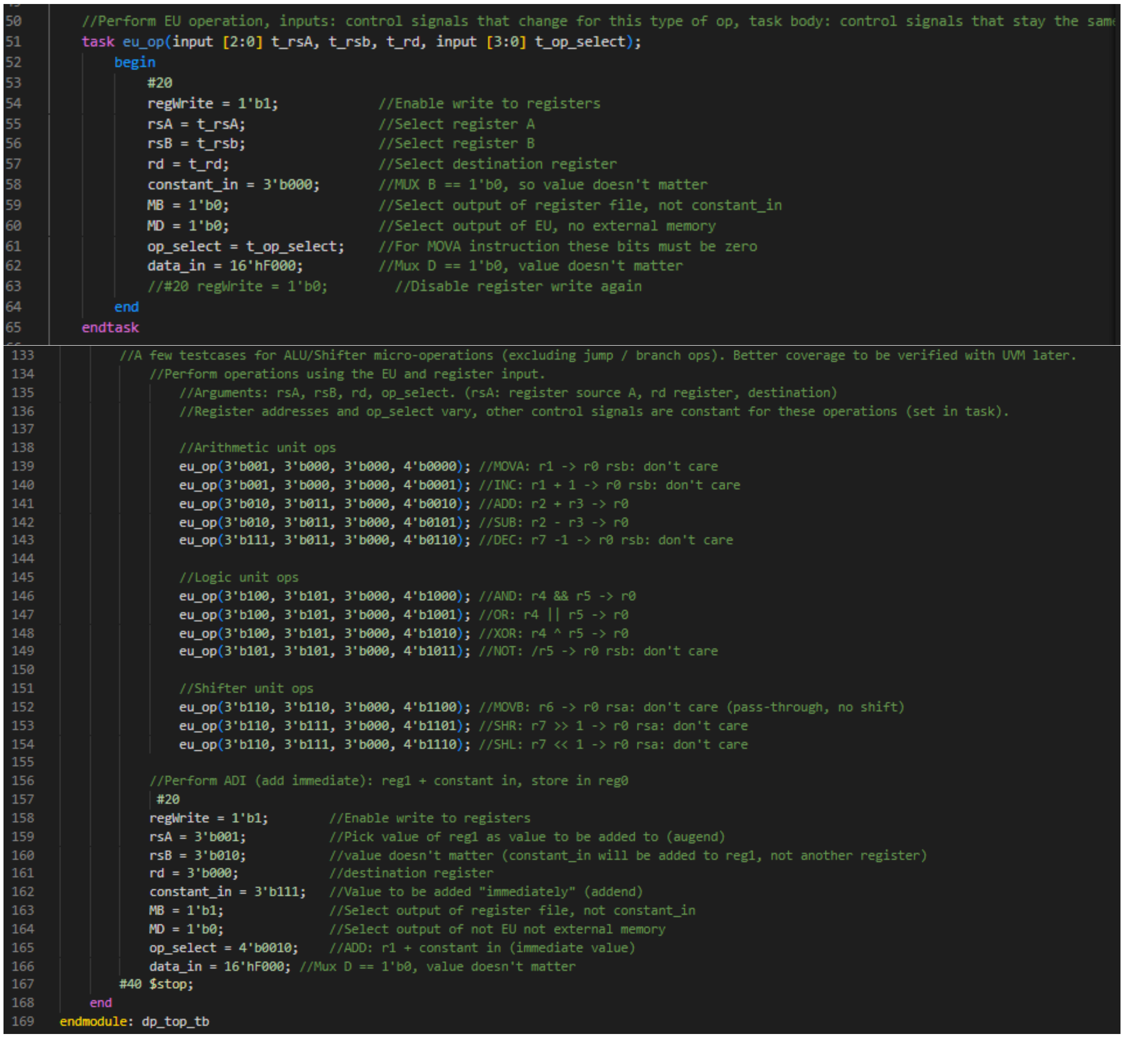


Figure 42 Figure 25 Test of the datapath top level, instruction execution



```
    //Perform EU operation, inputs: control signals that change for this type of op, task body: control signals that stay the same
    task eu_op(input [2:0] t_rsA, t_rsb, t_rd, input [3:0] t_op_select);
        begin
            #20
            regWrite = 1'b1;            //Enable write to registers
            rsA = t_rsA;                //Select register A
            rsB = t_rsb;                //Select register B
            rd = t_rd;                  //Select destination register
            constant_in = 3'b000;       //MUX B == 1'b0, so value doesn't matter
            MB = 1'b0;                  //Select output of register file, not constant_in
            MD = 1'b0;                  //Select output of EU, no external memory
            op_select = t_op_select;    //For MOVA instruction these bits must be zero
            data_in = 16'hF000;         //Mux D == 1'b0, value doesn't matter
            //#20 regWrite = 1'b0;        //Disable register write again
        end
    endtask
```

```
        //A few testcases for ALU/Shifter micro-operations (excluding jump / branch ops). Better coverage to be verified with UVM later.
            //Perform operations using the EU and register input.
                //Arguments: rsA, rsB, rd, op_select. (rsA: register source A, rd register, destination)
                //Register addresses and op_select vary, other control signals are constant for these operations (set in task).

                //Arithmetic unit ops
                eu_op(3'b001, 3'b000, 3'b000, 4'b0000); //MOVA: r1 -> r0 rsb: don't care
                eu_op(3'b001, 3'b000, 3'b000, 4'b0001); //INC: r1 + 1 -> r0 rsb: don't care
                eu_op(3'b010, 3'b011, 3'b000, 4'b0010); //ADD: r2 + r3 -> r0
                eu_op(3'b010, 3'b011, 3'b000, 4'b0101); //SUB: r2 - r3 -> r0
                eu_op(3'b111, 3'b011, 3'b000, 4'b0110); //DEC: r7 -1 -> r0 rsb: don't care

                //Logic unit ops
                eu_op(3'b100, 3'b101, 3'b000, 4'b1000); //AND: r4 && r5 -> r0
                eu_op(3'b100, 3'b101, 3'b000, 4'b1001); //OR: r4 || r5 -> r0
                eu_op(3'b100, 3'b101, 3'b000, 4'b1010); //XOR: r4 ^ r5 -> r0
                eu_op(3'b101, 3'b101, 3'b000, 4'b1011); //NOT: /r5 -> r0 rsb: don't care

                //Shifter unit ops
                eu_op(3'b110, 3'b110, 3'b000, 4'b1100); //MOVB: r6 -> r0 rsa: don't care (pass-through, no shift)
                eu_op(3'b110, 3'b111, 3'b000, 4'b1101); //SHR: r7 >> 1 -> r0 rsa: don't care
                eu_op(3'b110, 3'b111, 3'b000, 4'b1110); //SHL: r7 << 1 -> r0 rsa: don't care

            //Perform ADI (add immediate): reg1 + constant in, store in reg0
             #20
            regWrite = 1'b1;        //Enable write to registers
            rsA = 3'b001;           //Pick value of reg1 as value to be added to (augend)
            rsB = 3'b010;           //value doesn't matter (constant_in will be added to reg1, not another register)
            rd = 3'b000;            //destination register
            constant_in = 3'b111;   //Value to be added "immediately" (addend)
            MB = 1'b1;              //Select output of register file, not constant_in
            MD = 1'b0;              //Select output of not EU not external memory
            op_select = 4'b0010;    //ADD: r1 + constant in (immediate value)
            data_in = 16'hF000; //Mux D == 1'b0, value doesn't matter
        #40 $stop;
    end
endmodule: dp_top_tb
```

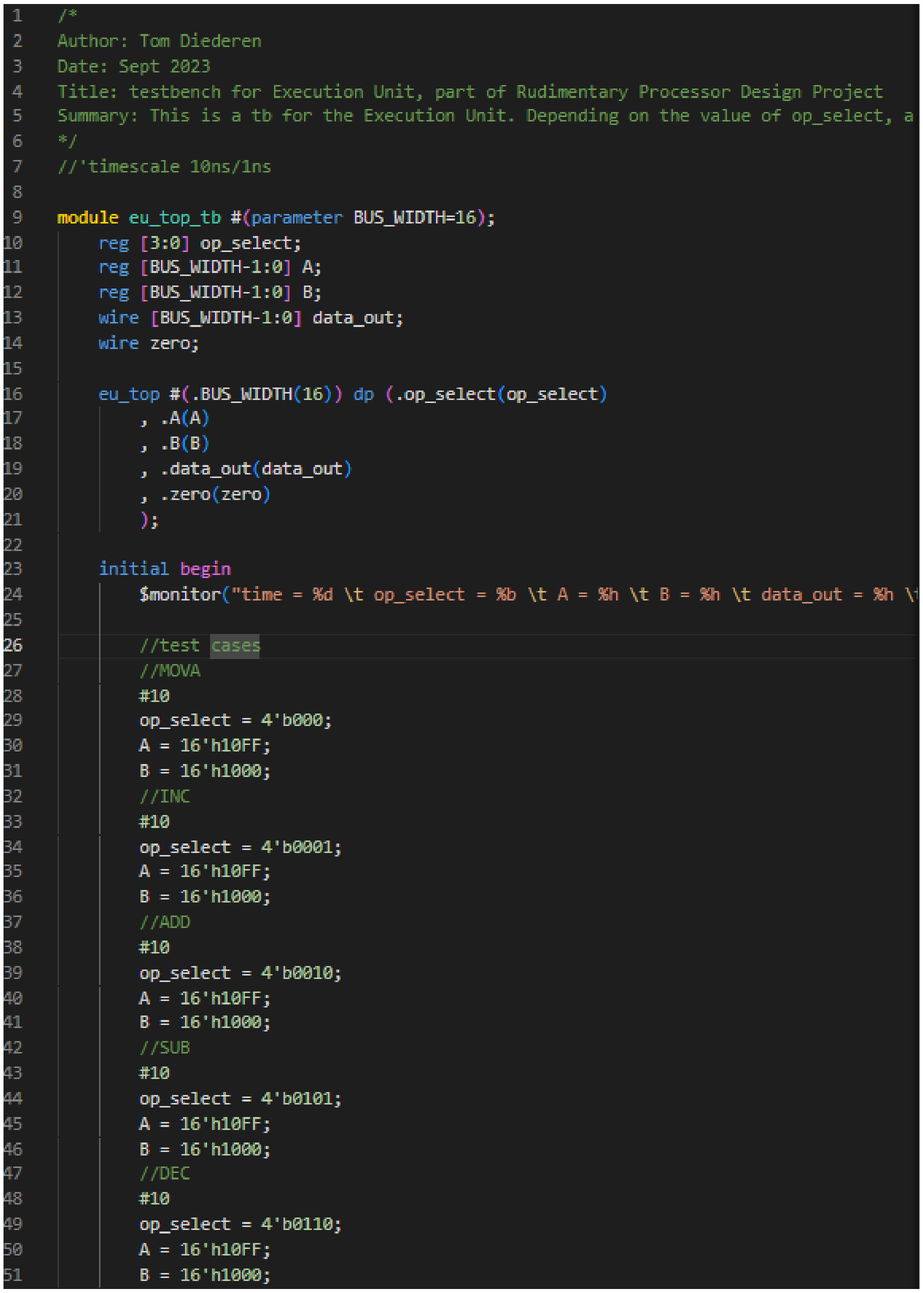
Figure 43 Verilog code for the datapath's top level testbench: instruction execution.

5.2.1.2 Execution Unit

Test 1: Perform All EU Operations

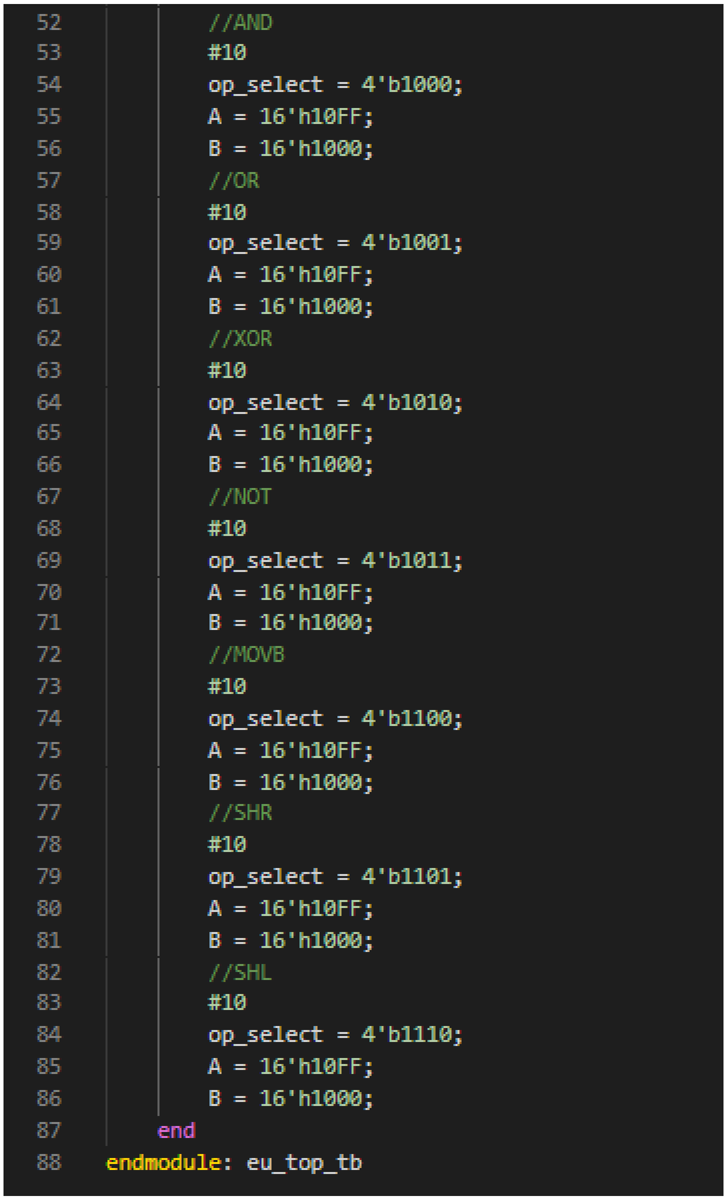
The top-level test performs each instruction for a randomly picked value for input A and B. Each unit, i.e. arithmetic, logic, and shifter, were tested individually before integration and running this top level test. These individual tests are described in the next sections.

Result: passed.



```
/*
Author: Tom Diederen
Date: Sept 2023
Title: testbench for Execution Unit, part of Rudimentary Processor Design Project
Summary: This is a tb for the Execution Unit. Depending on the value of op_select, a
*/
//'timescale 10ns/1ns

module eu_top_tb #(parameter BUS_WIDTH=16);
    reg [3:0] op_select;
    reg [BUS_WIDTH-1:0] A;
    reg [BUS_WIDTH-1:0] B;
    wire [BUS_WIDTH-1:0] data_out;
    wire zero;

    eu_top #(.BUS_WIDTH(16)) dp (.op_select(op_select)
        , .A(A)
        , .B(B)
        , .data_out(data_out)
        , .zero(zero)
        );

    initial begin
        $monitor("time = %d \t op_select = %b \t A = %h \t B = %h \t data_out = %h \t

        //test cases
        //MOVA
        #10
        op_select = 4'b000;
        A = 16'h10FF;
        B = 16'h1000;
        //INC
        #10
        op_select = 4'b0001;
        A = 16'h10FF;
        B = 16'h1000;
        //ADD
        #10
        op_select = 4'b0010;
        A = 16'h10FF;
        B = 16'h1000;
        //SUB
        #10
        op_select = 4'b0101;
        A = 16'h10FF;
        B = 16'h1000;
        //DEC
        #10
        op_select = 4'b0110;
        A = 16'h10FF;
        B = 16'h1000;
```

Figure 44 Top level test bench for the Execution Unit



```
        //AND
        #10
        op_select = 4'b1000;
        A = 16'h10FF;
        B = 16'h1000;
        //OR
        #10
        op_select = 4'b1001;
        A = 16'h10FF;
        B = 16'h1000;
        //XOR
        #10
        op_select = 4'b1010;
        A = 16'h10FF;
        B = 16'h1000;
        //NOT
        #10
        op_select = 4'b1011;
        A = 16'h10FF;
        B = 16'h1000;
        //MOVB
        #10
        op_select = 4'b1100;
        A = 16'h10FF;
        B = 16'h1000;
        //SHR
        #10
        op_select = 4'b1101;
        A = 16'h10FF;
        B = 16'h1000;
        //SHL
        #10
        op_select = 4'b1110;
        A = 16'h10FF;
        B = 16'h1000;
    end
endmodule: eu_top_tb
```

Figure 45 Top level test bench for the Execution Unit (continued)

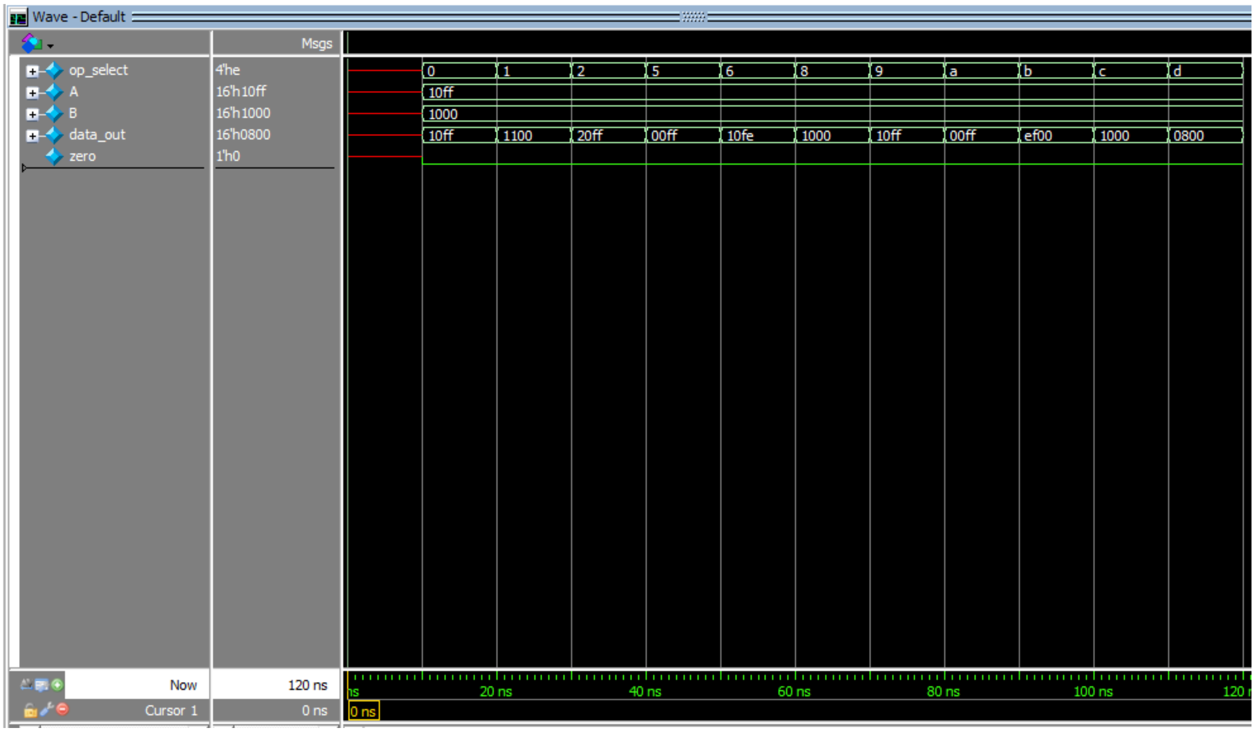
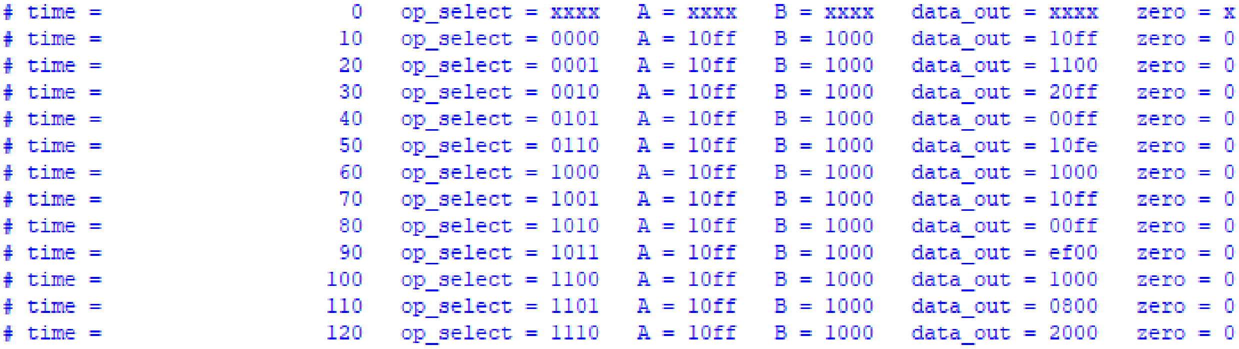


Figure 46 Simulation results for the Execution Unit (waveform)



```
# time =                    0   op_select = xxxx   A = xxxx   B = xxxx   data_out = xxxx   zero = x
# time =                   10   op_select = 0000   A = 10ff   B = 1000   data_out = 10ff   zero = 0
# time =                   20   op_select = 0001   A = 10ff   B = 1000   data_out = 1100   zero = 0
# time =                   30   op_select = 0010   A = 10ff   B = 1000   data_out = 20ff   zero = 0
# time =                   40   op_select = 0101   A = 10ff   B = 1000   data_out = 00ff   zero = 0
# time =                   50   op_select = 0110   A = 10ff   B = 1000   data_out = 10fe   zero = 0
# time =                   60   op_select = 1000   A = 10ff   B = 1000   data_out = 1000   zero = 0
# time =                   70   op_select = 1001   A = 10ff   B = 1000   data_out = 10ff   zero = 0
# time =                   80   op_select = 1010   A = 10ff   B = 1000   data_out = 00ff   zero = 0
# time =                   90   op_select = 1011   A = 10ff   B = 1000   data_out = ef00   zero = 0
# time =                  100   op_select = 1100   A = 10ff   B = 1000   data_out = 1000   zero = 0
# time =                  110   op_select = 1101   A = 10ff   B = 1000   data_out = 0800   zero = 0
# time =                  120   op_select = 1110   A = 10ff   B = 1000   data_out = 2000   zero = 0
```

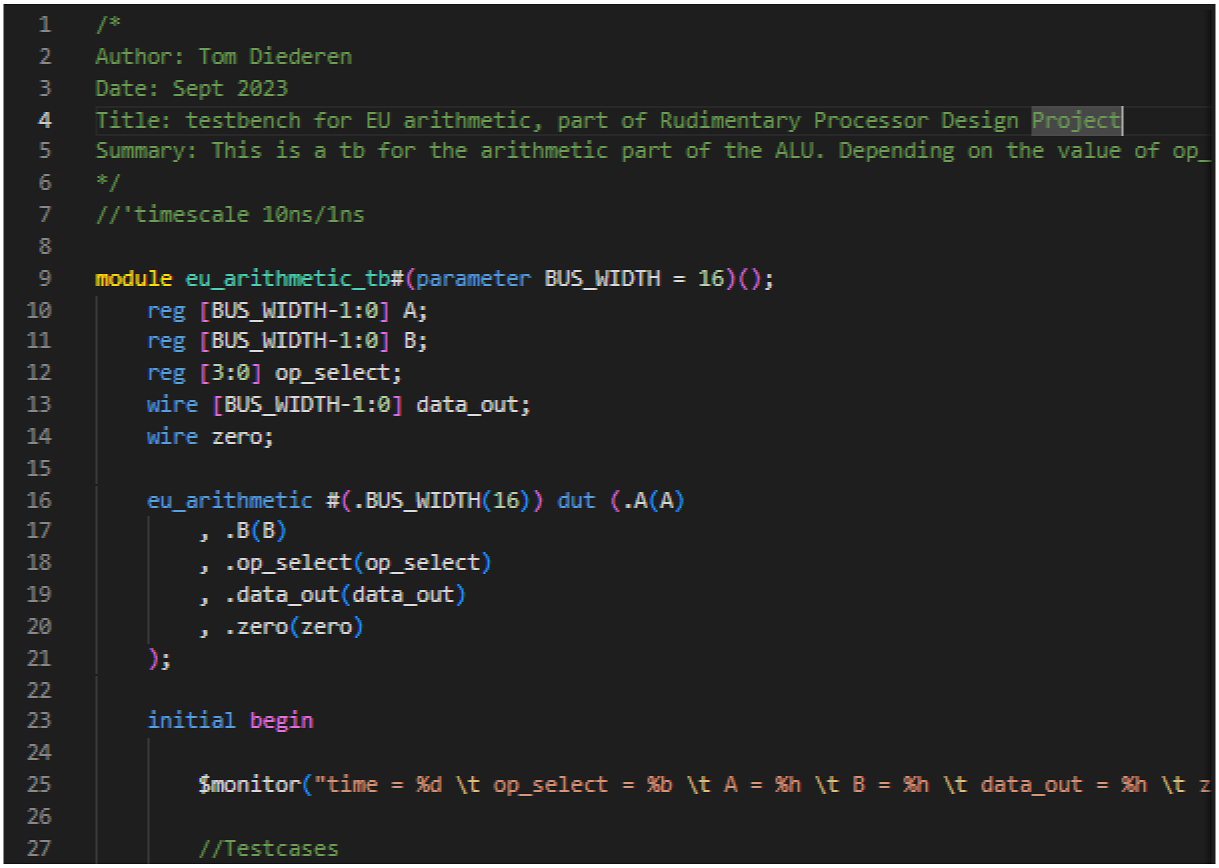
Figure 47 Simulation results for the Execution Unit (text)

Arithmetic Unit

Test 1: Perform Arithmetic Instructions

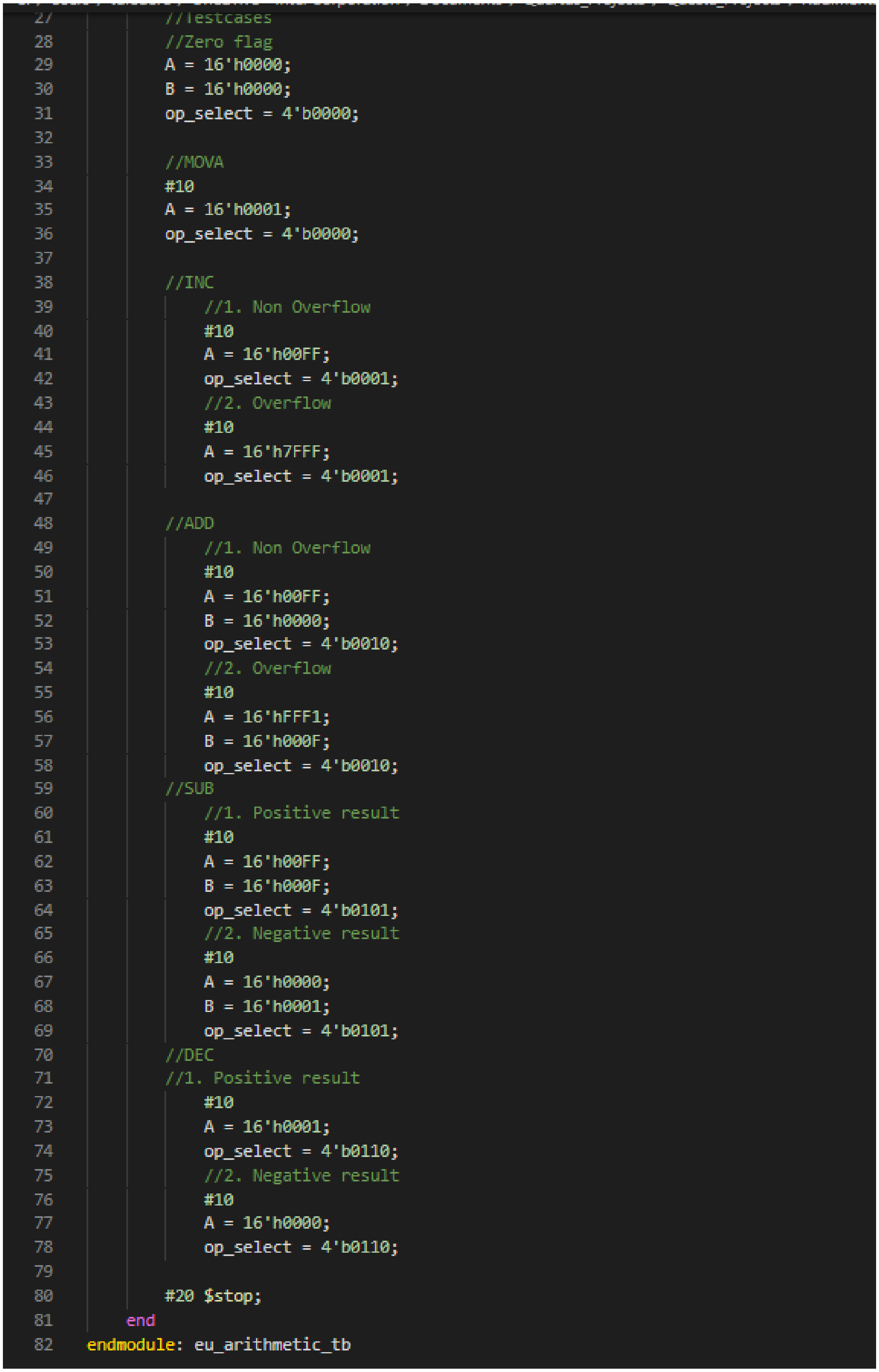
The testbench and results for the individual test of the arithmetic unit are shown below. As stated in the requirements section, overflow handling is not in scope for this project.

Result: passed.



```
/*
Author: Tom Diederen
Date: Sept 2023
Title: testbench for EU arithmetic, part of Rudimentary Processor Design Project
Summary: This is a tb for the arithmetic part of the ALU. Depending on the value of op_
*/
//'timescale 10ns/1ns

module eu_arithmetic_tb#(parameter BUS_WIDTH = 16)();
    reg [BUS_WIDTH-1:0] A;
    reg [BUS_WIDTH-1:0] B;
    reg [3:0] op_select;
    wire [BUS_WIDTH-1:0] data_out;
    wire zero;

    eu_arithmetic #(.BUS_WIDTH(16)) dut (.A(A)
        , .B(B)
        , .op_select(op_select)
        , .data_out(data_out)
        , .zero(zero)
    );

    initial begin

        $monitor("time = %d \t op_select = %b \t A = %h \t B = %h \t data_out = %h \t z

        //Testcases
```

Figure 48 RTL of the unit test bench for the Arithmetic Unit.



```
        //Testcases
        //Zero flag
        A = 16'h0000;
        B = 16'h0000;
        op_select = 4'b0000;

        //MOVA
        #10
        A = 16'h0001;
        op_select = 4'b0000;

        //INC
            //1. Non Overflow
            #10
            A = 16'h00FF;
            op_select = 4'b0001;
            //2. Overflow
            #10
            A = 16'h7FFF;
            op_select = 4'b0001;

        //ADD
            //1. Non Overflow
            #10
            A = 16'h00FF;
            B = 16'h0000;
            op_select = 4'b0010;
            //2. Overflow
            #10
            A = 16'hFFF1;
            B = 16'h000F;
            op_select = 4'b0010;
        //SUB
            //1. Positive result
            #10
            A = 16'h00FF;
            B = 16'h000F;
            op_select = 4'b0101;
            //2. Negative result
            #10
            A = 16'h0000;
            B = 16'h0001;
            op_select = 4'b0101;
        //DEC
        //1. Positive result
            #10
            A = 16'h0001;
            op_select = 4'b0110;
            //2. Negative result
            #10
            A = 16'h0000;
            op_select = 4'b0110;

        #20 $stop;
    end
endmodule: eu_arithmetic_tb
```

Figure 49 RTL of the unit test bench for the Arithmetic Unit (continued).

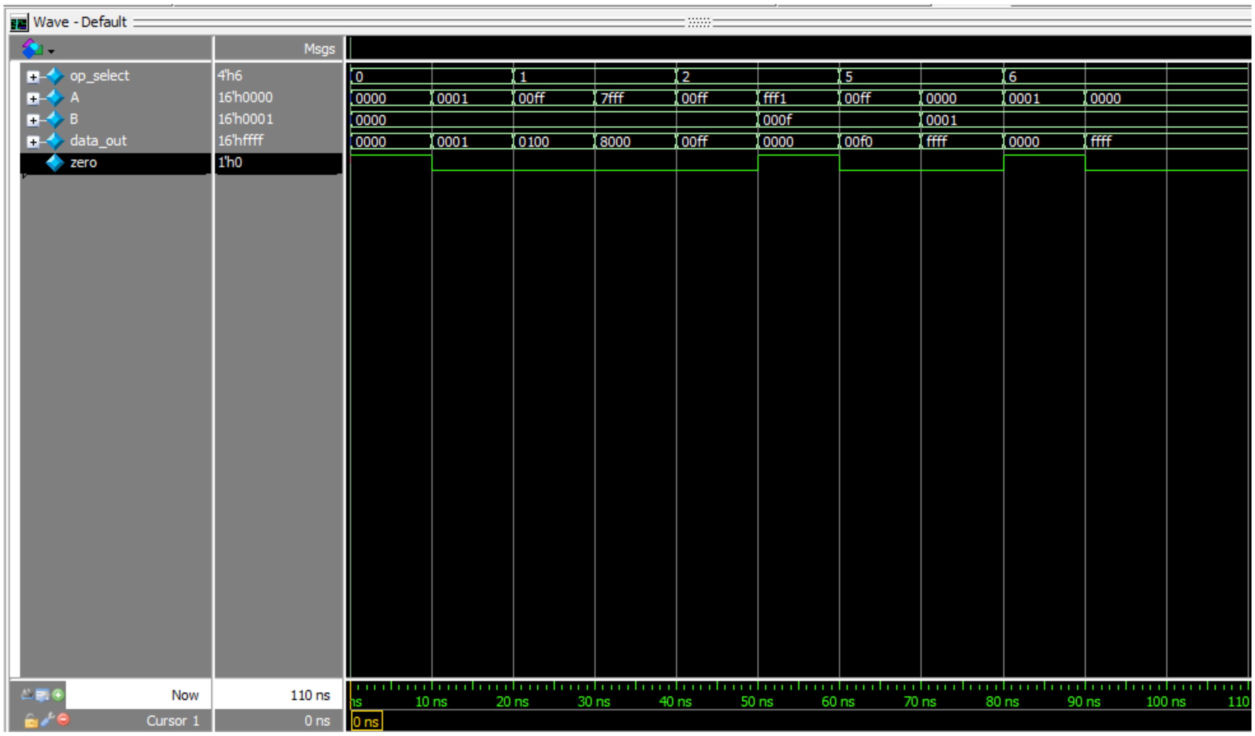
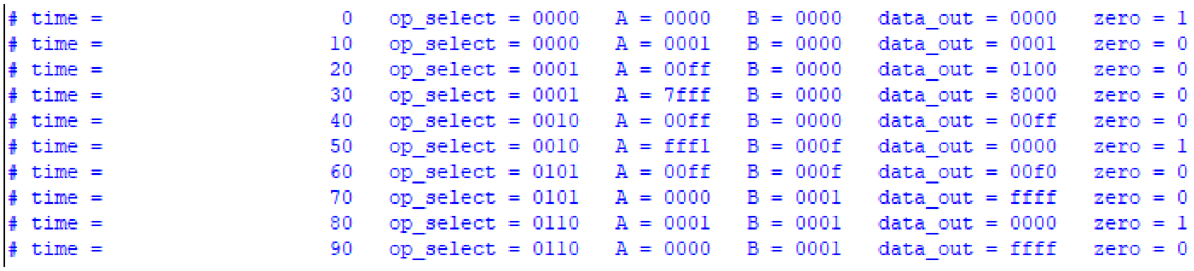

Wave - Default
Msgs
op_select 4'h6
A 16'h0000
B 16'h0001
data_out 16'hffff
zero 1'h0
0 1 2 5 6
0000 0001 00ff 7fff 00ff fff1 00ff 0000 0001 0000
0000 000f 0001
0000 0001 0100 8000 00ff 0000 00f0 ffff 0000 ffff
Now 110 ns
Cursor 1 0 ns
ns 10 ns 20 ns 30 ns 40 ns 50 ns 60 ns 70 ns 80 ns 90 ns 100 ns 110
0 ns

Figure 50 Simulation results for the Arithmetic Unit (waveform)



```
# time =                     0   op_select = 0000   A = 0000   B = 0000   data_out = 0000   zero = 1
# time =                    10   op_select = 0000   A = 0001   B = 0000   data_out = 0001   zero = 0
# time =                    20   op_select = 0001   A = 00ff   B = 0000   data_out = 0100   zero = 0
# time =                    30   op_select = 0001   A = 7fff   B = 0000   data_out = 8000   zero = 0
# time =                    40   op_select = 0010   A = 00ff   B = 0000   data_out = 00ff   zero = 0
# time =                    50   op_select = 0010   A = fff1   B = 000f   data_out = 0000   zero = 1
# time =                    60   op_select = 0101   A = 00ff   B = 000f   data_out = 00f0   zero = 0
# time =                    70   op_select = 0101   A = 0000   B = 0001   data_out = ffff   zero = 0
# time =                    80   op_select = 0110   A = 0001   B = 0001   data_out = 0000   zero = 1
# time =                    90   op_select = 0110   A = 0000   B = 0001   data_out = ffff   zero = 0
```

Figure 51 Simulation results for the Arithmetic Unit (text)

Logic Unit

Test 1: Perform Logic Instructions

The logic unit testbench has a similar design compared to the one of the arithmetic unit shown above. It checks each logic operation of the unit for correctness.

Result: passed.

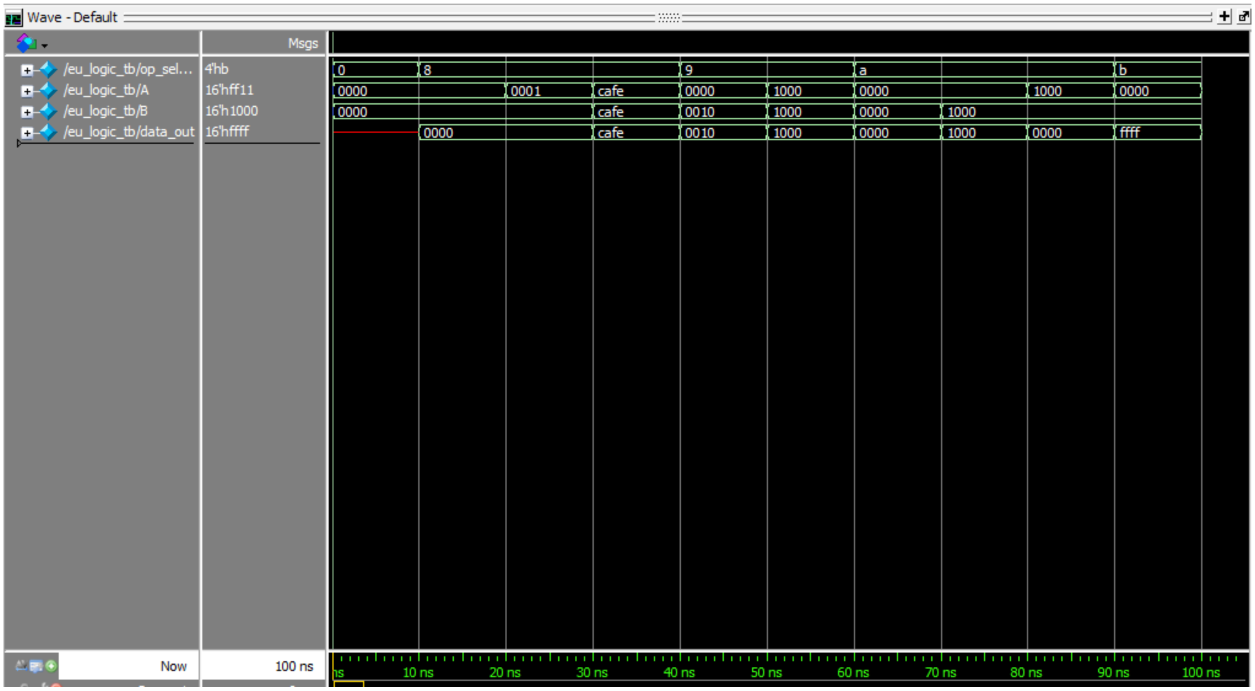
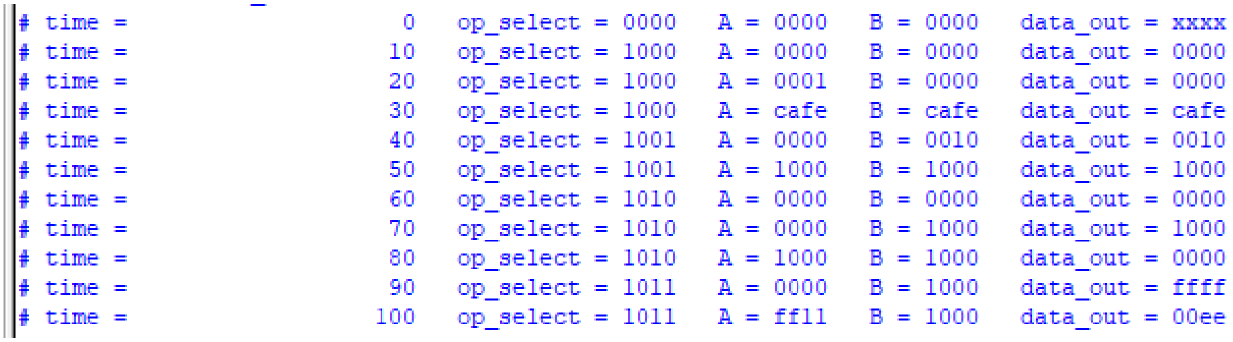
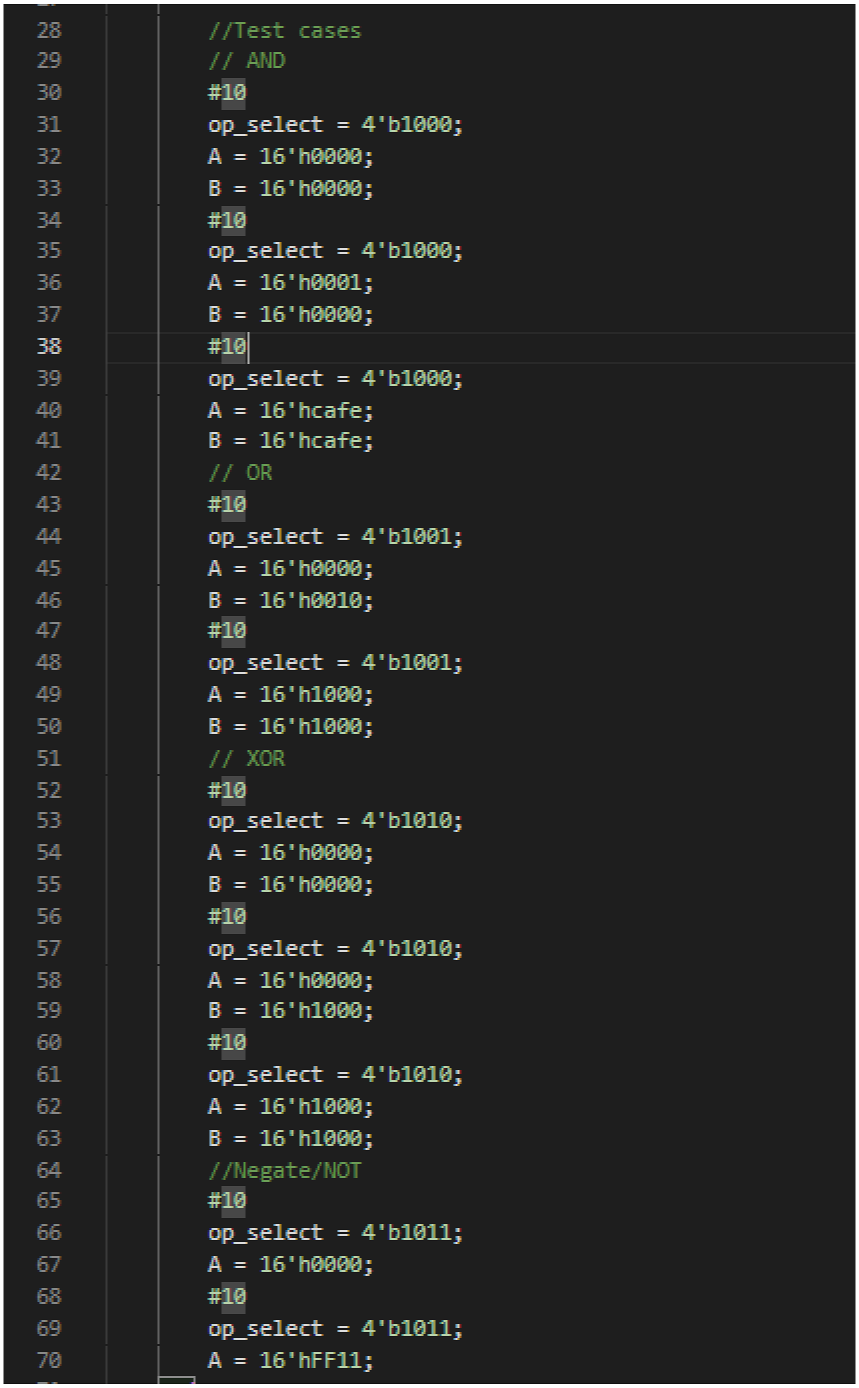


Figure 52 Simulation results for the Logic Unit (waveform)



```
# time =                0  op_select = 0000  A = 0000  B = 0000  data_out = xxxx
# time =               10  op_select = 1000  A = 0000  B = 0000  data_out = 0000
# time =               20  op_select = 1000  A = 0001  B = 0000  data_out = 0000
# time =               30  op_select = 1000  A = cafe  B = cafe  data_out = cafe
# time =               40  op_select = 1001  A = 0000  B = 0010  data_out = 0010
# time =               50  op_select = 1001  A = 1000  B = 1000  data_out = 1000
# time =               60  op_select = 1010  A = 0000  B = 0000  data_out = 0000
# time =               70  op_select = 1010  A = 0000  B = 1000  data_out = 1000
# time =               80  op_select = 1010  A = 1000  B = 1000  data_out = 0000
# time =               90  op_select = 1011  A = 0000  B = 1000  data_out = ffff
# time =              100  op_select = 1011  A = ff11  B = 1000  data_out = 00ee
```

Figure 53 Simulation results for the Logic Unit (text)



```
        //Test cases
        // AND
        #10
        op_select = 4'b1000;
        A = 16'h0000;
        B = 16'h0000;
        #10
        op_select = 4'b1000;
        A = 16'h0001;
        B = 16'h0000;
        #10
        op_select = 4'b1000;
        A = 16'hcafe;
        B = 16'hcafe;
        // OR
        #10
        op_select = 4'b1001;
        A = 16'h0000;
        B = 16'h0010;
        #10
        op_select = 4'b1001;
        A = 16'h1000;
        B = 16'h1000;
        // XOR
        #10
        op_select = 4'b1010;
        A = 16'h0000;
        B = 16'h0000;
        #10
        op_select = 4'b1010;
        A = 16'h0000;
        B = 16'h1000;
        #10
        op_select = 4'b1010;
        A = 16'h1000;
        B = 16'h1000;
        //Negate/NOT
        #10
        op_select = 4'b1011;
        A = 16'h0000;
        #10
        op_select = 4'b1011;
        A = 16'hFF11;
```

Figure 54 Test cases for the logic testbench (Verilog)

Shifter

Test 1: Perform Shift Instructions

The testbench approach for the shifter is similar to the one taken for the arithmetic and logic parts.

Result: passed.

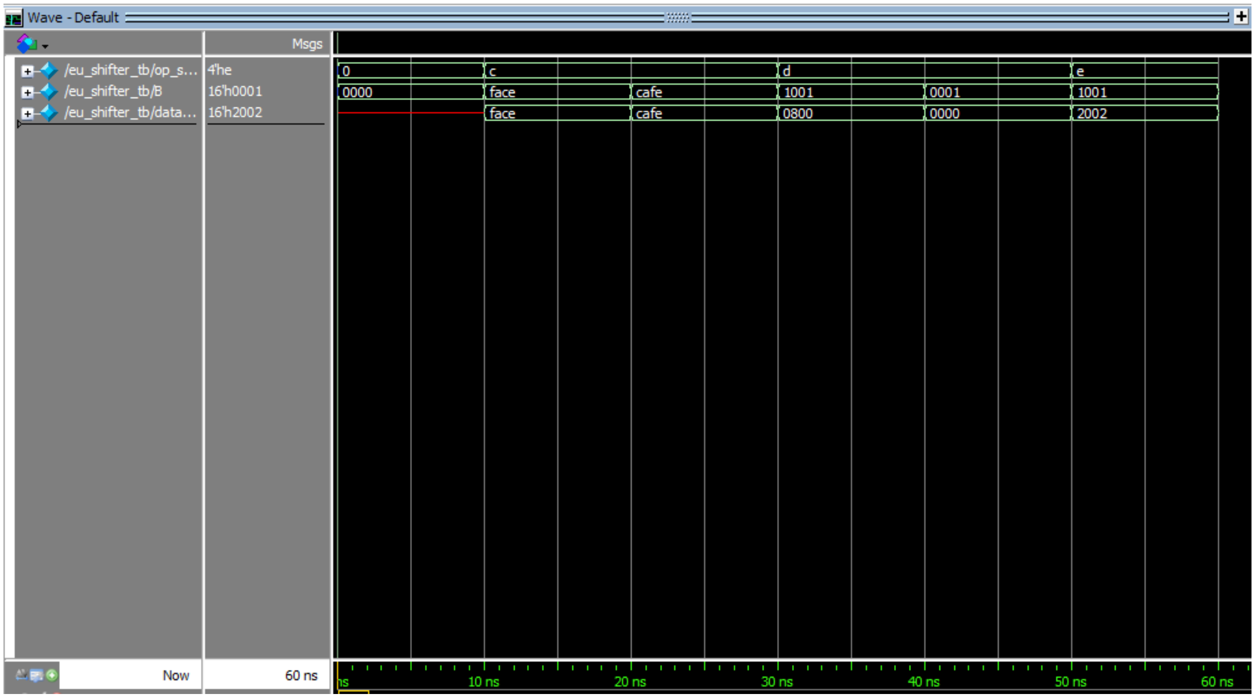
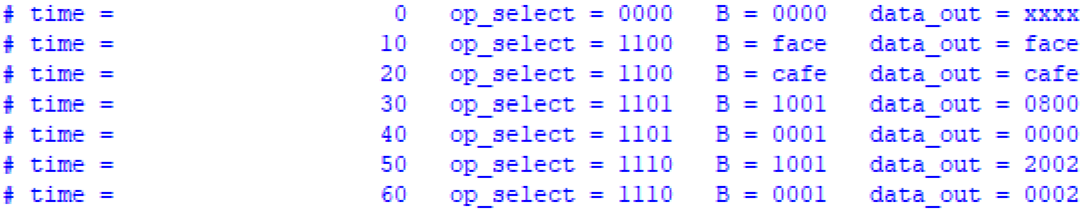
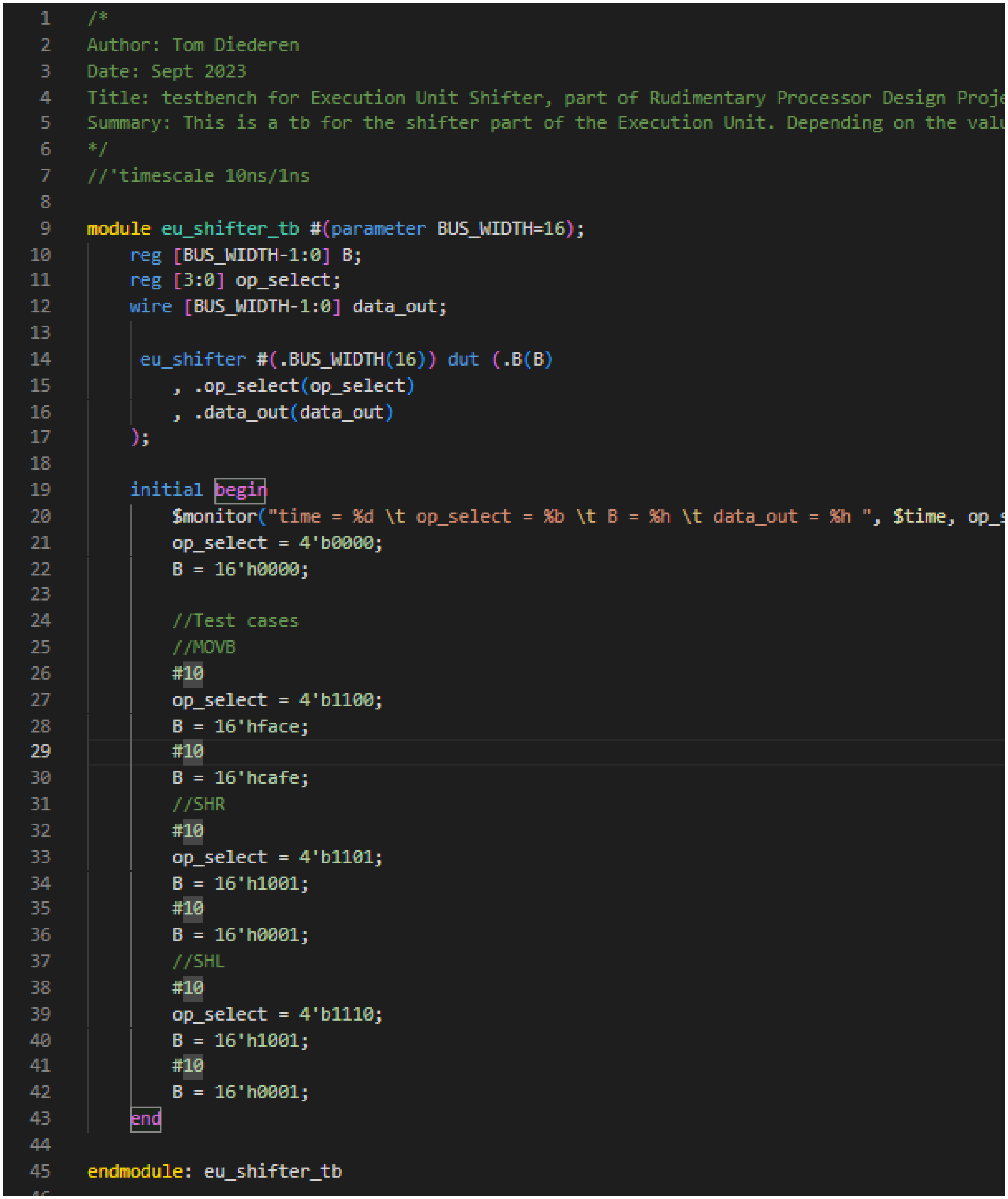


Figure 55 Simulation results for the shifter (waveform)



```
# time =                    0   op_select = 0000   B = 0000   data_out = xxxx
# time =                   10   op_select = 1100   B = face   data_out = face
# time =                   20   op_select = 1100   B = cafe   data_out = cafe
# time =                   30   op_select = 1101   B = 1001   data_out = 0800
# time =                   40   op_select = 1101   B = 0001   data_out = 0000
# time =                   50   op_select = 1110   B = 1001   data_out = 2002
# time =                   60   op_select = 1110   B = 0001   data_out = 0002
```

Figure 56 Simulation results for the shifter (text)



```verilog
/*
Author: Tom Diederen
Date: Sept 2023
Title: testbench for Execution Unit Shifter, part of Rudimentary Processor Design Proje
Summary: This is a tb for the shifter part of the Execution Unit. Depending on the valu
*/
//'timescale 10ns/1ns

module eu_shifter_tb #(parameter BUS_WIDTH=16);
    reg [BUS_WIDTH-1:0] B;
    reg [3:0] op_select;
    wire [BUS_WIDTH-1:0] data_out;

     eu_shifter #(.BUS_WIDTH(16)) dut (.B(B)
        , .op_select(op_select)
        , .data_out(data_out)
    );

    initial begin
        $monitor("time = %d \t op_select = %b \t B = %h \t data_out = %h ", $time, op_s
        op_select = 4'b0000;
        B = 16'h0000;

        //Test cases
        //MOVB
        #10
        op_select = 4'b1100;
        B = 16'hface;
        #10
        B = 16'hcafe;
        //SHR
        #10
        op_select = 4'b1101;
        B = 16'h1001;
        #10
        B = 16'h0001;
        //SHL
        #10
        op_select = 4'b1110;
        B = 16'h1001;
        #10
        B = 16'h0001;
    end

endmodule: eu_shifter_tb
```

Figure 57 Shifter testbench (Verilog)

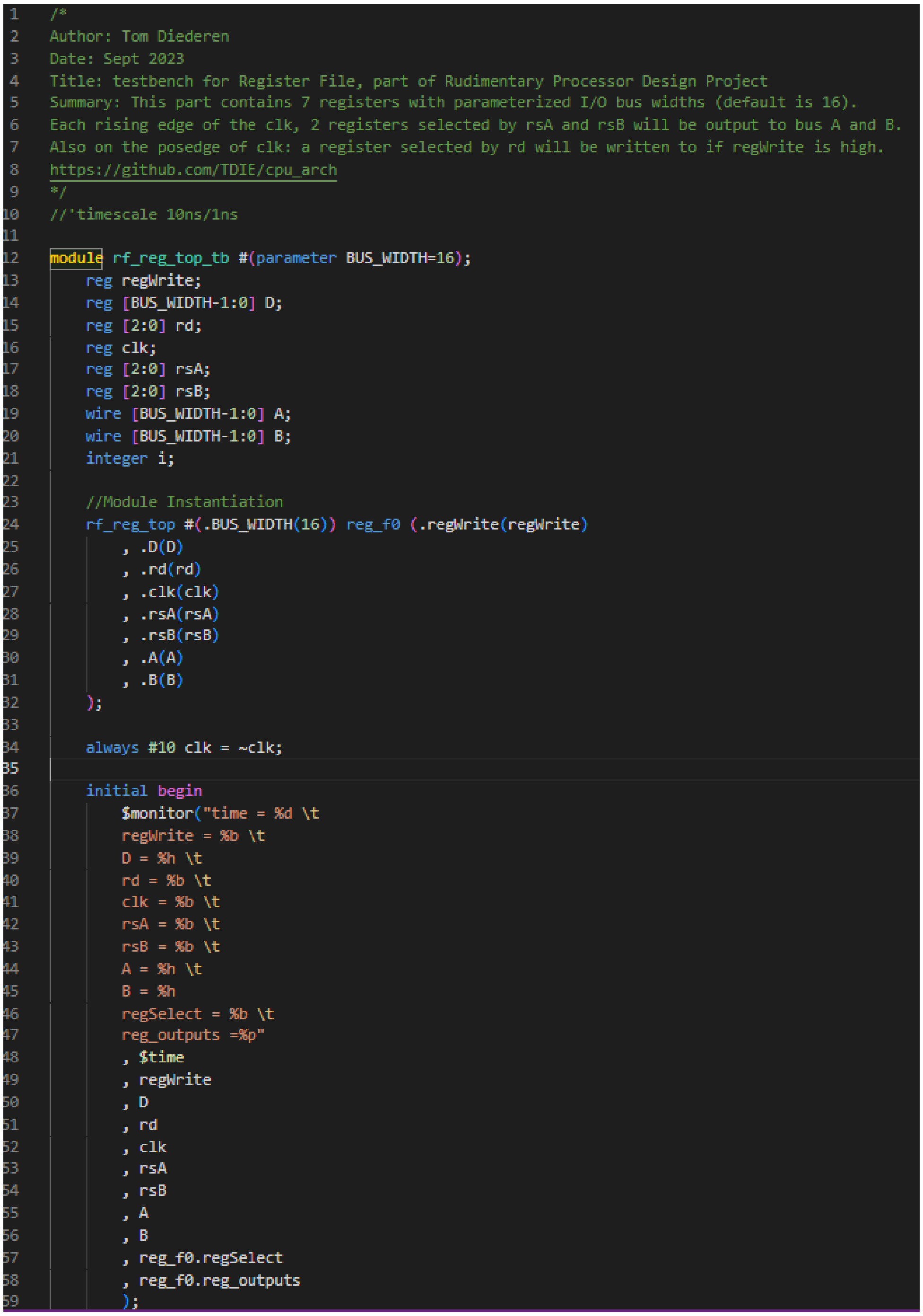
5.2.1.3 Register File

The unit tests for the register file are shown below. For reference, the top layer is shown first but the design was tested bottom up (the individual register was tested first, followed by 8 of them connected in the top layer)

Test 1: Load All Registers of the Register File

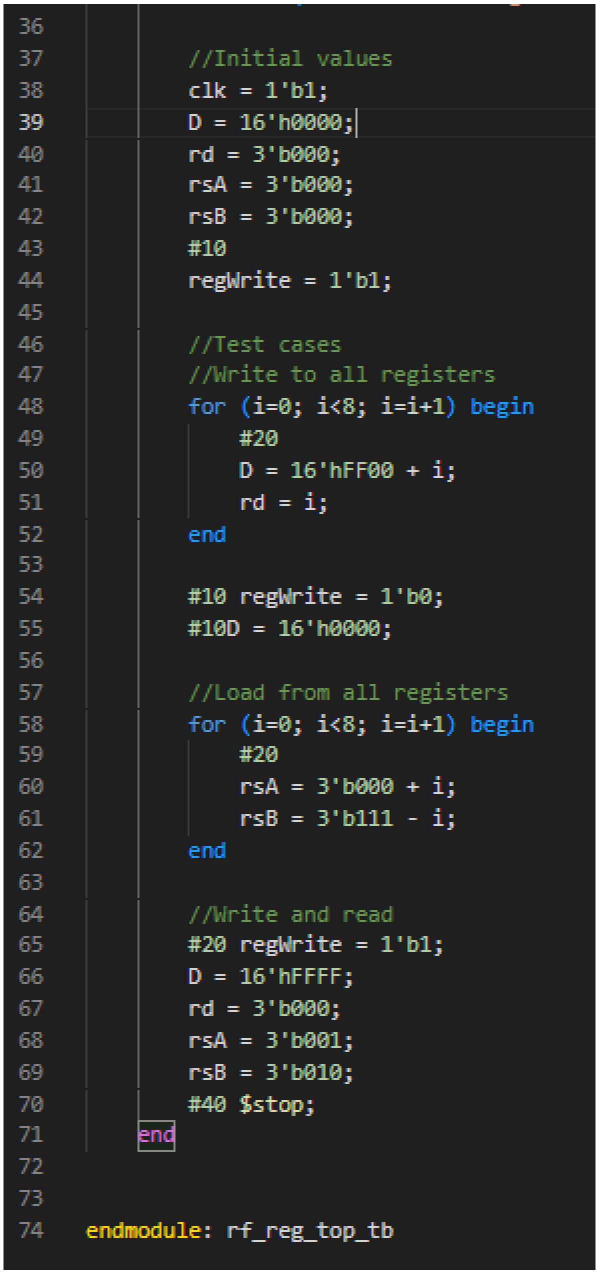
The testbench instantiates the Register File containing 8, 16-bit, registers. A monitor tracks all input and output values of the top layer as well as the values of regWrite and output of the individual registers. The test cases are as follows: write to all registers, read from all registers, read and write at the same time.

The Verilog description and results are shown below.



```verilog
/*
Author: Tom Diederen
Date: Sept 2023
Title: testbench for Register File, part of Rudimentary Processor Design Project
Summary: This part contains 7 registers with parameterized I/O bus widths (default is 16).
Each rising edge of the clk, 2 registers selected by rsA and rsB will be output to bus A and B.
Also on the posedge of clk: a register selected by rd will be written to if regWrite is high.
https://github.com/TDIE/cpu_arch
*/
//'timescale 10ns/1ns

module rf_reg_top_tb #(parameter BUS_WIDTH=16);
    reg regWrite;
    reg [BUS_WIDTH-1:0] D;
    reg [2:0] rd;
    reg clk;
    reg [2:0] rsA;
    reg [2:0] rsB;
    wire [BUS_WIDTH-1:0] A;
    wire [BUS_WIDTH-1:0] B;
    integer i;

    //Module Instantiation
    rf_reg_top #(.BUS_WIDTH(16)) reg_f0 (.regWrite(regWrite)
        , .D(D)
        , .rd(rd)
        , .clk(clk)
        , .rsA(rsA)
        , .rsB(rsB)
        , .A(A)
        , .B(B)
    );

    always #10 clk = ~clk;

    initial begin
        $monitor("time = %d \t
        regWrite = %b \t
        D = %h \t
        rd = %b \t
        clk = %b \t
        rsA = %b \t
        rsB = %b \t
        A = %h \t
        B = %h
        regSelect = %b \t
        reg_outputs =%p"
        , $time
        , regWrite
        , D
        , rd
        , clk
        , rsA
        , rsB
        , A
        , B
        , reg_f0.regSelect
        , reg_f0.reg_outputs
        );
```

Figure 58 Verilog description of the register file testbench: IO, module instantiation, and monitor.



```verilog

        //Initial values
        clk = 1'b1;
        D = 16'h0000;
        rd = 3'b000;
        rsA = 3'b000;
        rsB = 3'b000;
        #10
        regWrite = 1'b1;

        //Test cases
        //Write to all registers
        for (i=0; i<8; i=i+1) begin
            #20
            D = 16'hFF00 + i;
            rd = i;
        end

        #10 regWrite = 1'b0;
        #10D = 16'h0000;

        //Load from all registers
        for (i=0; i<8; i=i+1) begin
            #20
            rsA = 3'b000 + i;
            rsB = 3'b111 - i;
        end

        //Write and read
        #20 regWrite = 1'b1;
        D = 16'hFFFF;
        rd = 3'b000;
        rsA = 3'b001;
        rsB = 3'b010;
        #40 $stop;
    end


endmodule: rf_reg_top_tb
```

Figure 59 Verilog description of the testbench for the register file: test stimuli.

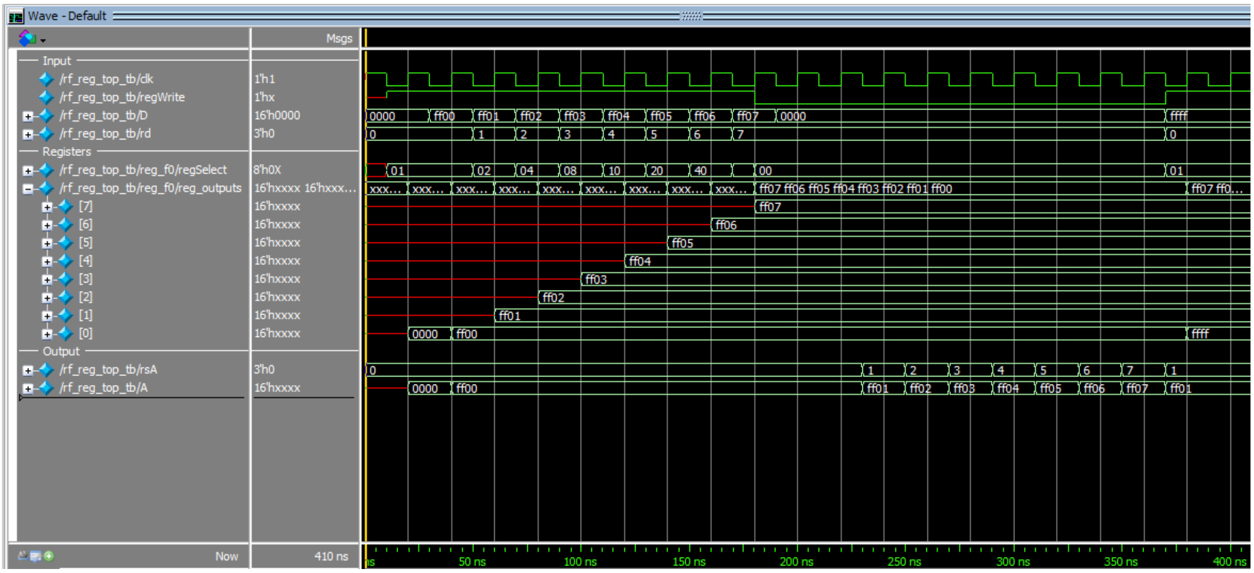


Figure 60 Test results for the top layer of the Register File (wave).

Test 2: Load Single Register

The test results for a single register are shown below. The testbench design is similar to the top level but simpler.

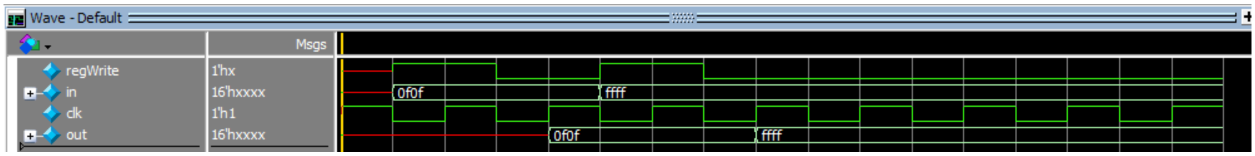
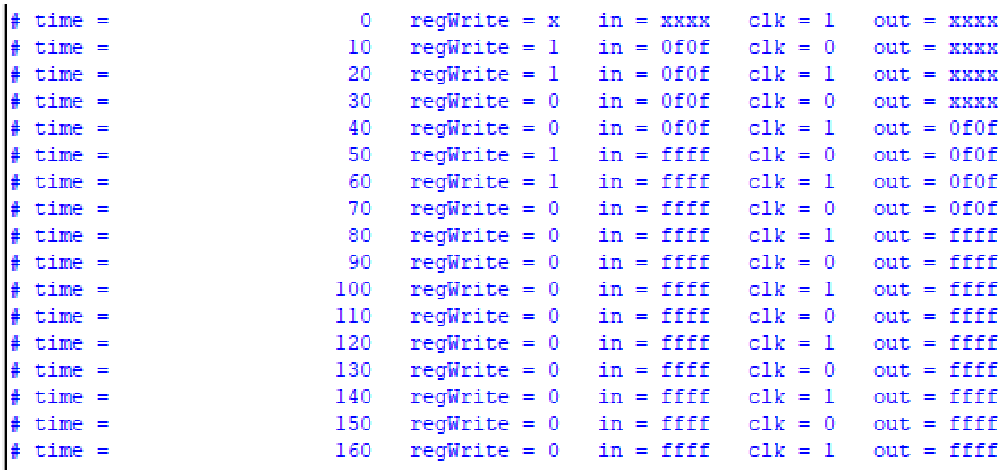
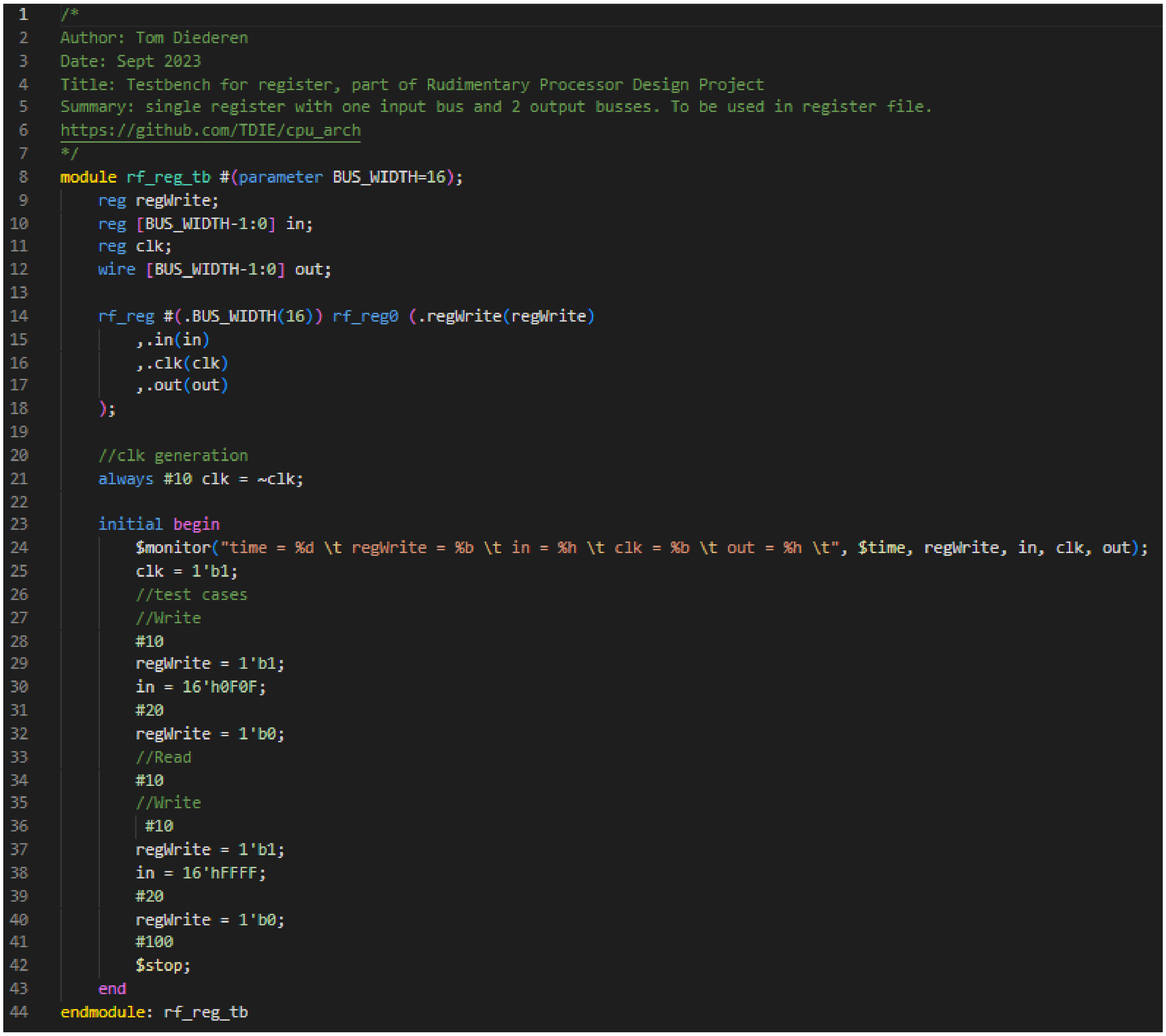


Figure 61 Testbench results for an individual register (wave)



```
# time =                    0   regWrite = x   in = xxxx   clk = 1   out = xxxx
# time =                   10   regWrite = 1   in = 0f0f   clk = 0   out = xxxx
# time =                   20   regWrite = 1   in = 0f0f   clk = 1   out = xxxx
# time =                   30   regWrite = 0   in = 0f0f   clk = 0   out = xxxx
# time =                   40   regWrite = 0   in = 0f0f   clk = 1   out = 0f0f
# time =                   50   regWrite = 1   in = ffff   clk = 0   out = 0f0f
# time =                   60   regWrite = 1   in = ffff   clk = 1   out = 0f0f
# time =                   70   regWrite = 0   in = ffff   clk = 0   out = 0f0f
# time =                   80   regWrite = 0   in = ffff   clk = 1   out = ffff
# time =                   90   regWrite = 0   in = ffff   clk = 0   out = ffff
# time =                  100   regWrite = 0   in = ffff   clk = 1   out = ffff
# time =                  110   regWrite = 0   in = ffff   clk = 0   out = ffff
# time =                  120   regWrite = 0   in = ffff   clk = 1   out = ffff
# time =                  130   regWrite = 0   in = ffff   clk = 0   out = ffff
# time =                  140   regWrite = 0   in = ffff   clk = 1   out = ffff
# time =                  150   regWrite = 0   in = ffff   clk = 0   out = ffff
# time =                  160   regWrite = 0   in = ffff   clk = 1   out = ffff
```

Figure 62 Testbench results for an individual register (text).



```
/*
Author: Tom Diederen
Date: Sept 2023
Title: Testbench for register, part of Rudimentary Processor Design Project
Summary: single register with one input bus and 2 output busses. To be used in register file.
https://github.com/TDIE/cpu_arch
*/
module rf_reg_tb #(parameter BUS_WIDTH=16);
    reg regWrite;
    reg [BUS_WIDTH-1:0] in;
    reg clk;
    wire [BUS_WIDTH-1:0] out;

    rf_reg #(.BUS_WIDTH(16)) rf_reg0 (.regWrite(regWrite)
        ,.in(in)
        ,.clk(clk)
        ,.out(out)
    );

    //clk generation
    always #10 clk = ~clk;

    initial begin
        $monitor("time = %d \t regWrite = %b \t in = %h \t clk = %b \t out = %h \t", $time, regWrite, in, clk, out);
        clk = 1'b1;
        //test cases
        //Write
        #10
        regWrite = 1'b1;
        in = 16'h0F0F;
        #20
        regWrite = 1'b0;
        //Read
        #10
        //Write
         #10
        regWrite = 1'b1;
        in = 16'hFFFF;
        #20
        regWrite = 1'b0;
        #100
        $stop;
    end
endmodule: rf_reg_tb
```

Figure 63 Verilog description of the testbench for a single register.

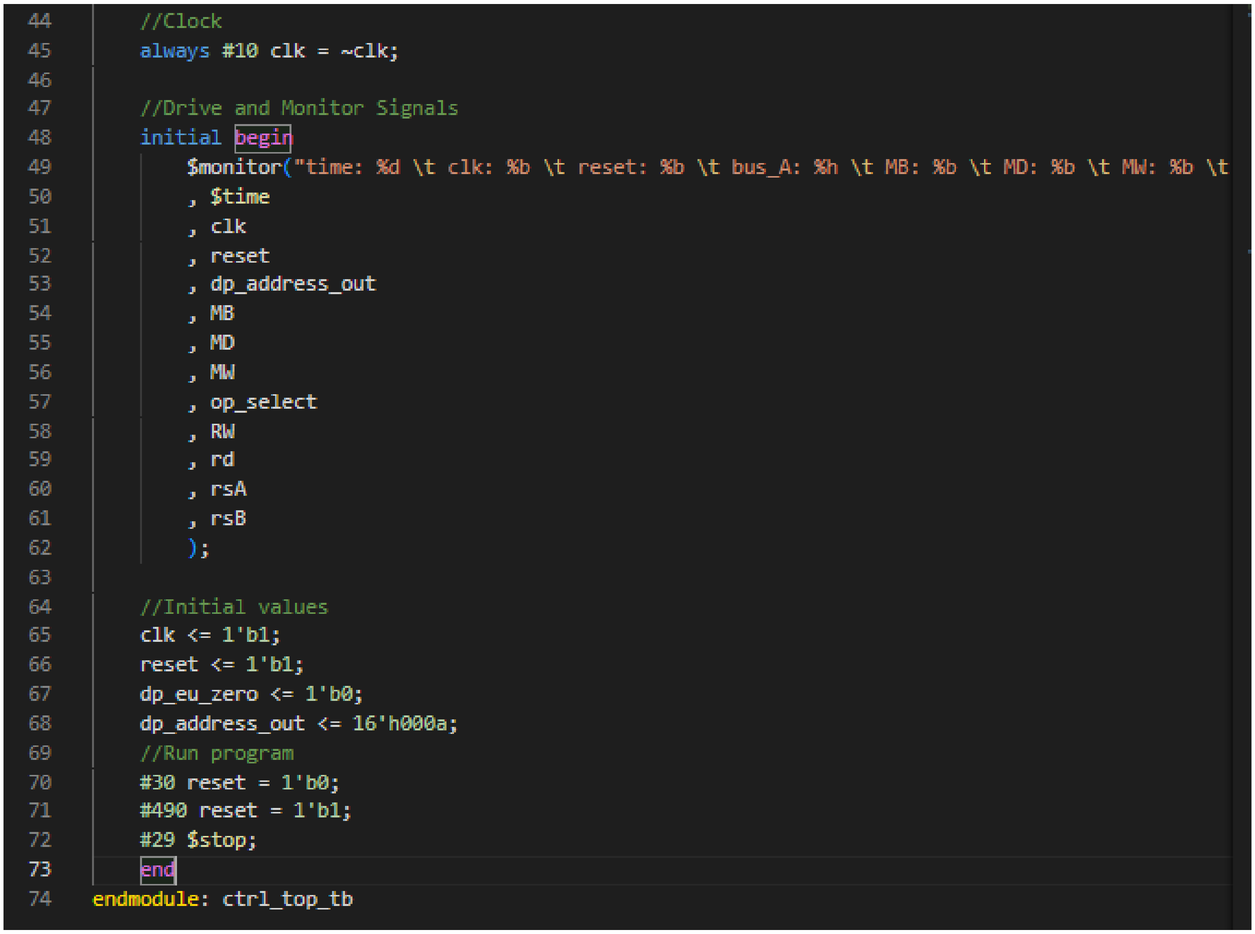
5.2.2 Control Unit

5.2.2.1 Test 1: Control Unit Output for a Sample Program

As described in section 3.1.2, the Control Unit consists of three components: the Program Counter, PC, Instruction Memory, and the Instruction Decoder. This section describes the test results for the complete Control Unit. Individual unit tests of the sub-components were performed earlier before integrating them into the Control Unit. These sub-component tests are shown in the next sections.

As described in section 3.1.2.2, the instruction memory holds a sample program that performs each instruction type once. An overview is included below for reference. Cross referencing these instructions with the instruction decoder, described in section 3.1.2.3, shows that the Control Unit works correctly.

Result: passed.



```
    //Clock
    always #10 clk = ~clk;

    //Drive and Monitor Signals
    initial begin
        $monitor("time: %d \t clk: %b \t reset: %b \t bus_A: %h \t MB: %b \t MD: %b \t MW: %b \t
        , $time
        , clk
        , reset
        , dp_address_out
        , MB
        , MD
        , MW
        , op_select
        , RW
        , rd
        , rsA
        , rsB
        );

    //Initial values
    clk <= 1'b1;
    reset <= 1'b1;
    dp_eu_zero <= 1'b0;
    dp_address_out <= 16'h000a;
    //Run program
    #30 reset = 1'b0;
    #490 reset = 1'b1;
    #29 $stop;
    end
endmodule: ctrl_top_tb
```

Figure 64 Verilog testbench for the Control Unit's top layer.

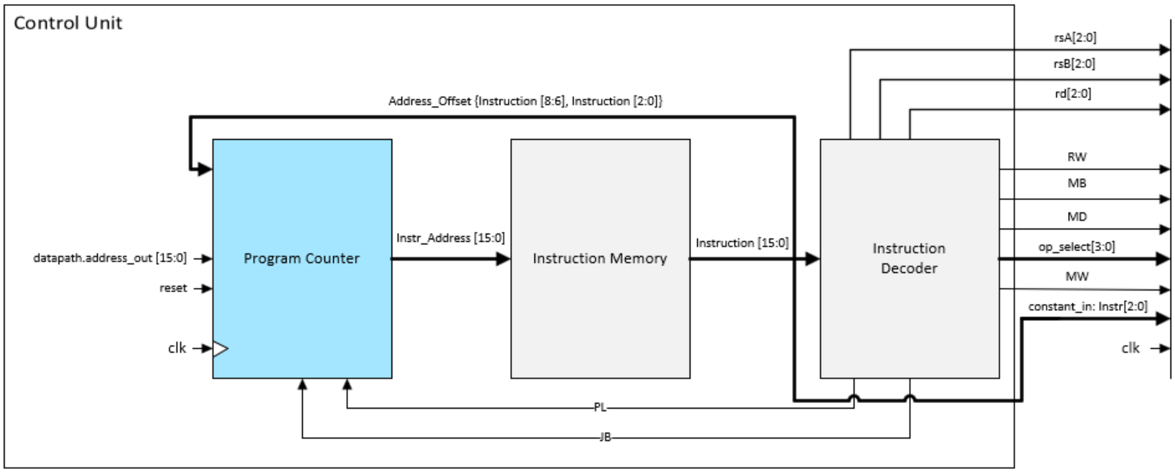


Figure 65 Overview of the Control Unit

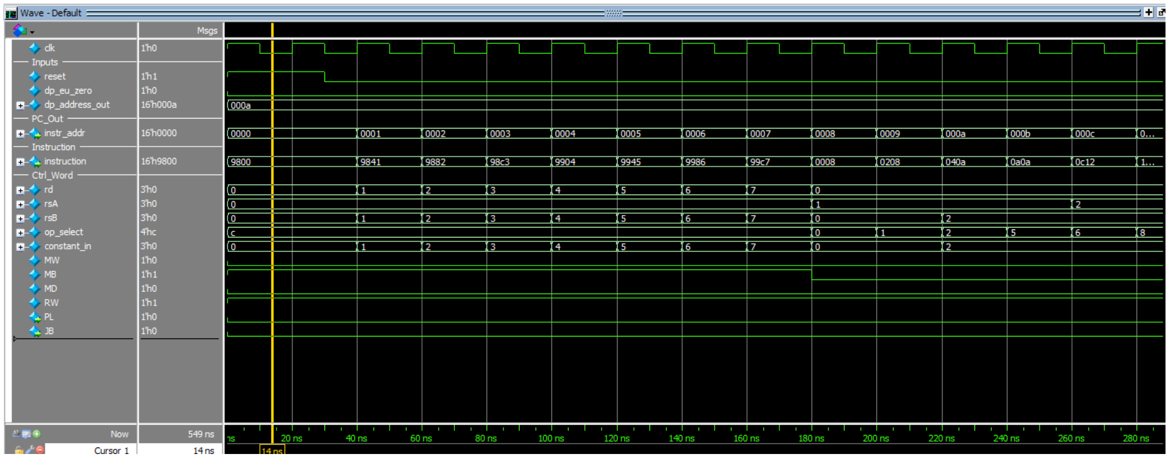


Figure 66 Testbench results of the Control Unit's top layer.

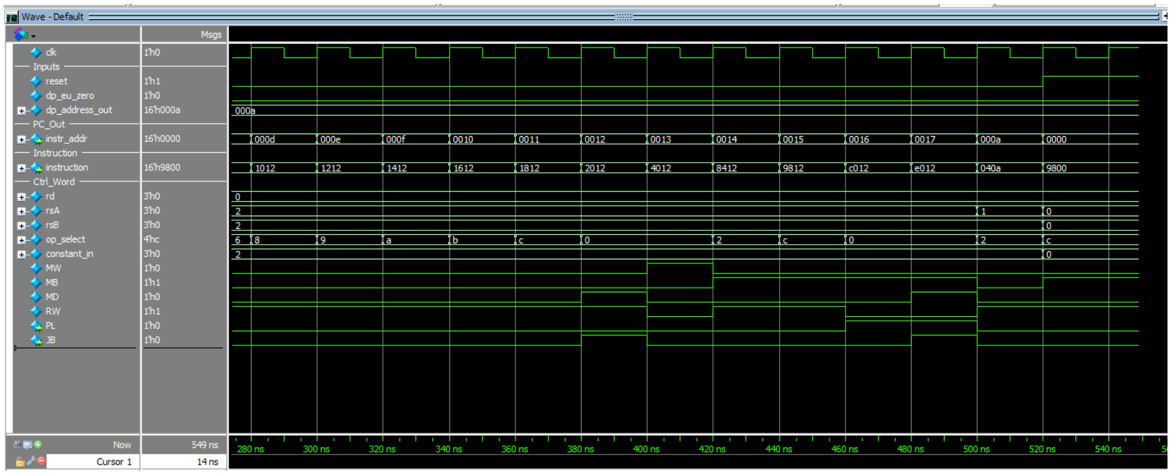
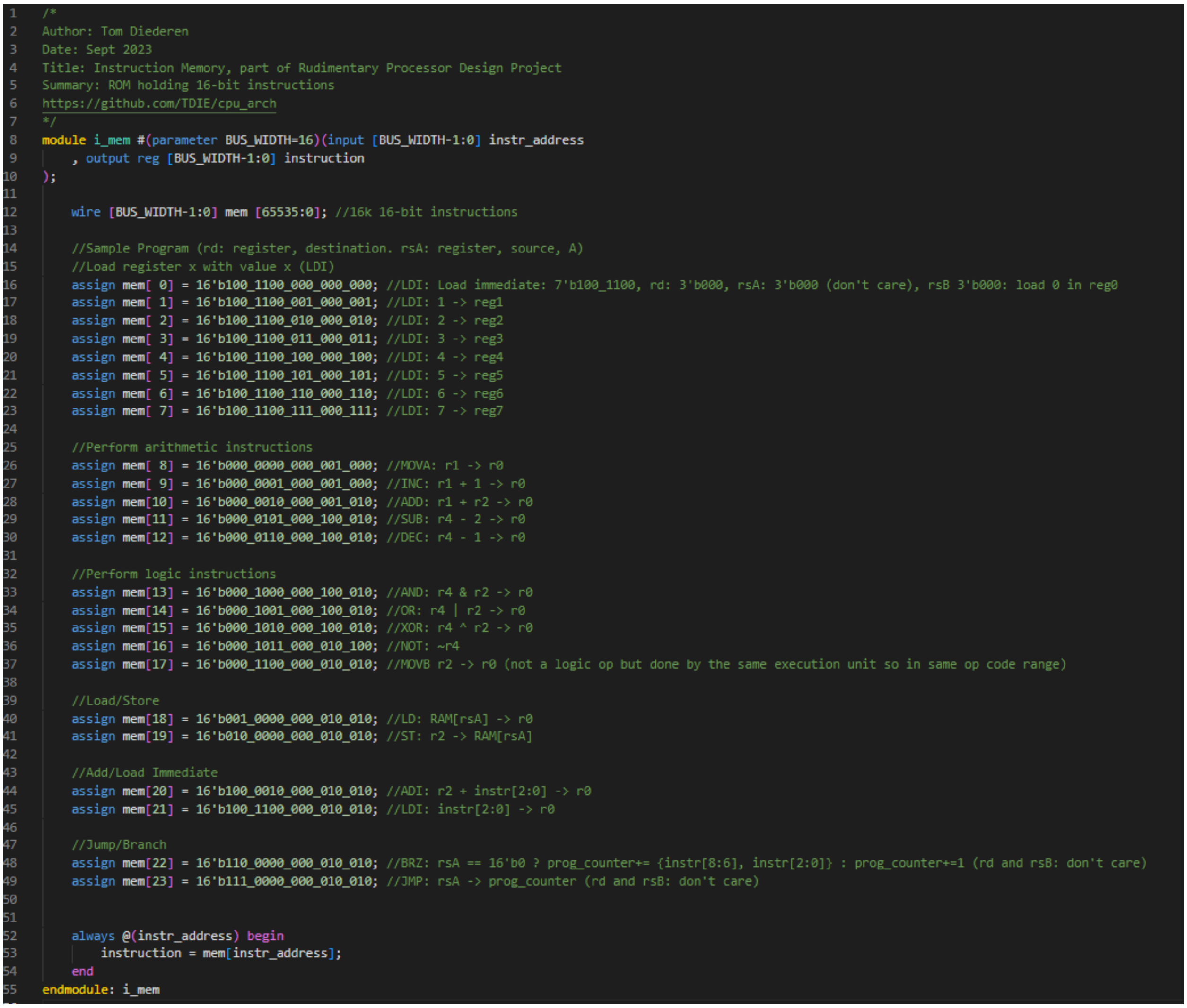


Figure 67 Testbench results of the Control Unit's top layer (continued).



```verilog
/*
Author: Tom Diederen
Date: Sept 2023
Title: Instruction Memory, part of Rudimentary Processor Design Project
Summary: ROM holding 16-bit instructions
https://github.com/TDIE/cpu_arch
*/
module i_mem #(parameter BUS_WIDTH=16)(input [BUS_WIDTH-1:0] instr_address
    , output reg [BUS_WIDTH-1:0] instruction
);

    wire [BUS_WIDTH-1:0] mem [65535:0]; //16k 16-bit instructions

    //Sample Program (rd: register, destination. rsA: register, source, A)
    //Load register x with value x (LDI)
    assign mem[ 0] = 16'b100_1100_000_000_000; //LDI: Load immediate: 7'b100_1100, rd: 3'b000, rsA: 3'b000 (don't care), rsB 3'b000: load 0 in reg0
    assign mem[ 1] = 16'b100_1100_001_000_001; //LDI: 1 -> reg1
    assign mem[ 2] = 16'b100_1100_010_000_010; //LDI: 2 -> reg2
    assign mem[ 3] = 16'b100_1100_011_000_011; //LDI: 3 -> reg3
    assign mem[ 4] = 16'b100_1100_100_000_100; //LDI: 4 -> reg4
    assign mem[ 5] = 16'b100_1100_101_000_101; //LDI: 5 -> reg5
    assign mem[ 6] = 16'b100_1100_110_000_110; //LDI: 6 -> reg6
    assign mem[ 7] = 16'b100_1100_111_000_111; //LDI: 7 -> reg7

    //Perform arithmetic instructions
    assign mem[ 8] = 16'b000_0000_000_001_000; //MOVA: r1 -> r0
    assign mem[ 9] = 16'b000_0001_000_001_000; //INC: r1 + 1 -> r0
    assign mem[10] = 16'b000_0010_000_001_010; //ADD: r1 + r2 -> r0
    assign mem[11] = 16'b000_0101_000_100_010; //SUB: r4 - 2 -> r0
    assign mem[12] = 16'b000_0110_000_100_010; //DEC: r4 - 1 -> r0

    //Perform logic instructions
    assign mem[13] = 16'b000_1000_000_100_010; //AND: r4 & r2 -> r0
    assign mem[14] = 16'b000_1001_000_100_010; //OR: r4 | r2 -> r0
    assign mem[15] = 16'b000_1010_000_100_010; //XOR: r4 ^ r2 -> r0
    assign mem[16] = 16'b000_1011_000_010_100; //NOT: ~r4
    assign mem[17] = 16'b000_1100_000_010_010; //MOVB r2 -> r0 (not a logic op but done by the same execution unit so in same op code range)

    //Load/Store
    assign mem[18] = 16'b001_0000_000_010_010; //LD: RAM[rsA] -> r0
    assign mem[19] = 16'b010_0000_000_010_010; //ST: r2 -> RAM[rsA]

    //Add/Load Immediate
    assign mem[20] = 16'b100_0010_000_010_010; //ADI: r2 + instr[2:0] -> r0
    assign mem[21] = 16'b100_1100_000_010_010; //LDI: instr[2:0] -> r0

    //Jump/Branch
    assign mem[22] = 16'b110_0000_000_010_010; //BRZ: rsA == 16'b0 ? prog_counter+= {instr[8:6], instr[2:0]} : prog_counter+=1 (rd and rsB: don't care)
    assign mem[23] = 16'b111_0000_000_010_010; //JMP: rsA -> prog_counter (rd and rsB: don't care)


    always @(instr_address) begin
        instruction = mem[instr_address];
    end
endmodule: i_mem
```

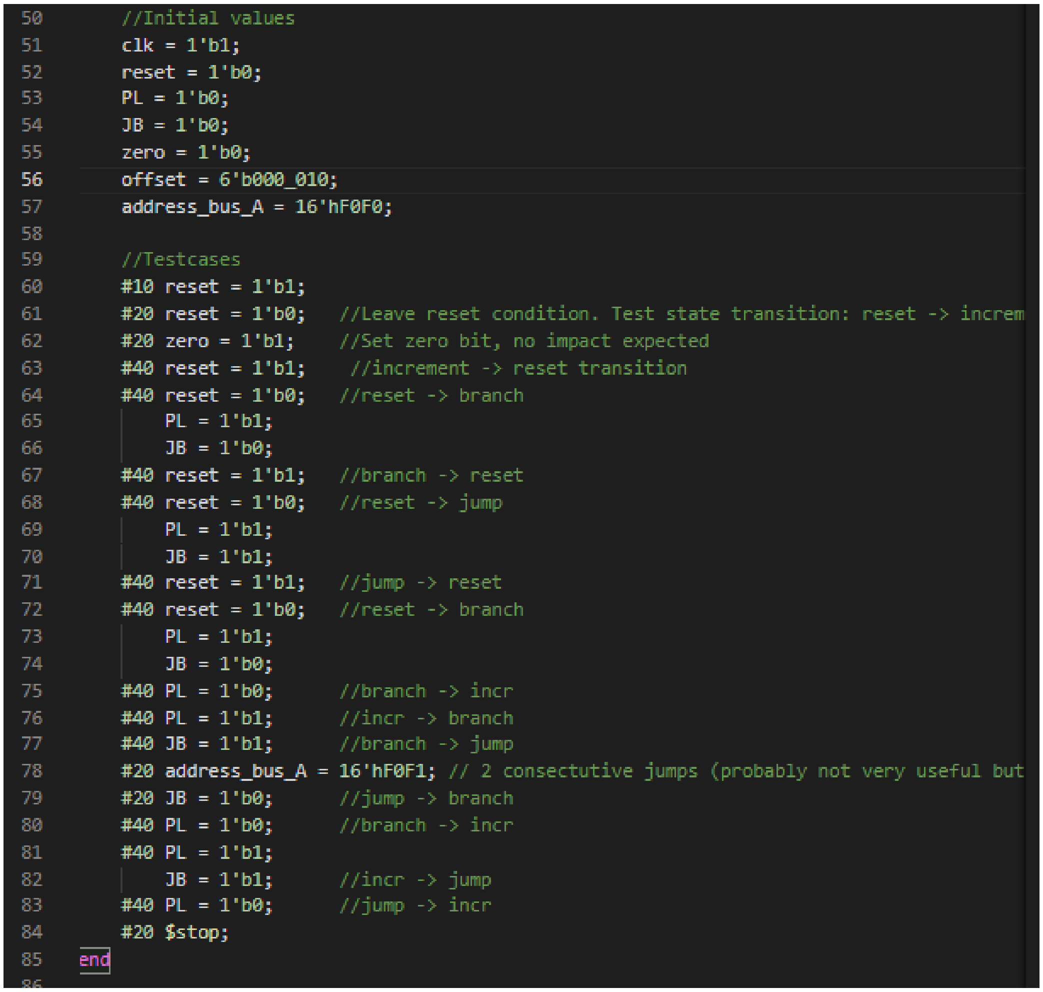
Figure 68 The sample program in the instruction memory.

5.2.2.2 Program Counter

Test 1: Cover All State Transitions

The Program Counter, described in section 3.1.2.1, contains a state machine that determines the next address for the instruction memory. The test below covers all state transitions.

Result: passed.



```verilog
    //Initial values
    clk = 1'b1;
    reset = 1'b0;
    PL = 1'b0;
    JB = 1'b0;
    zero = 1'b0;
    offset = 6'b000_010;
    address_bus_A = 16'hF0F0;

    //Testcases
    #10 reset = 1'b1;
    #20 reset = 1'b0;   //Leave reset condition. Test state transition: reset -> increm
    #20 zero = 1'b1;    //Set zero bit, no impact expected
    #40 reset = 1'b1;    //increment -> reset transition
    #40 reset = 1'b0;   //reset -> branch
        PL = 1'b1;
        JB = 1'b0;
    #40 reset = 1'b1;   //branch -> reset
    #40 reset = 1'b0;   //reset -> jump
        PL = 1'b1;
        JB = 1'b1;
    #40 reset = 1'b1;   //jump -> reset
    #40 reset = 1'b0;   //reset -> branch
        PL = 1'b1;
        JB = 1'b0;
    #40 PL = 1'b0;      //branch -> incr
    #40 PL = 1'b1;      //incr -> branch
    #40 JB = 1'b1;      //branch -> jump
    #20 address_bus_A = 16'hF0F1; // 2 consectutive jumps (probably not very useful but
    #20 JB = 1'b0;      //jump -> branch
    #40 PL = 1'b0;      //branch -> incr
    #40 PL = 1'b1;
        JB = 1'b1;      //incr -> jump
    #40 PL = 1'b0;      //jump -> incr
    #20 $stop;
end
```

Figure 69 Testcases for the Program Counter (Verilog testbench)

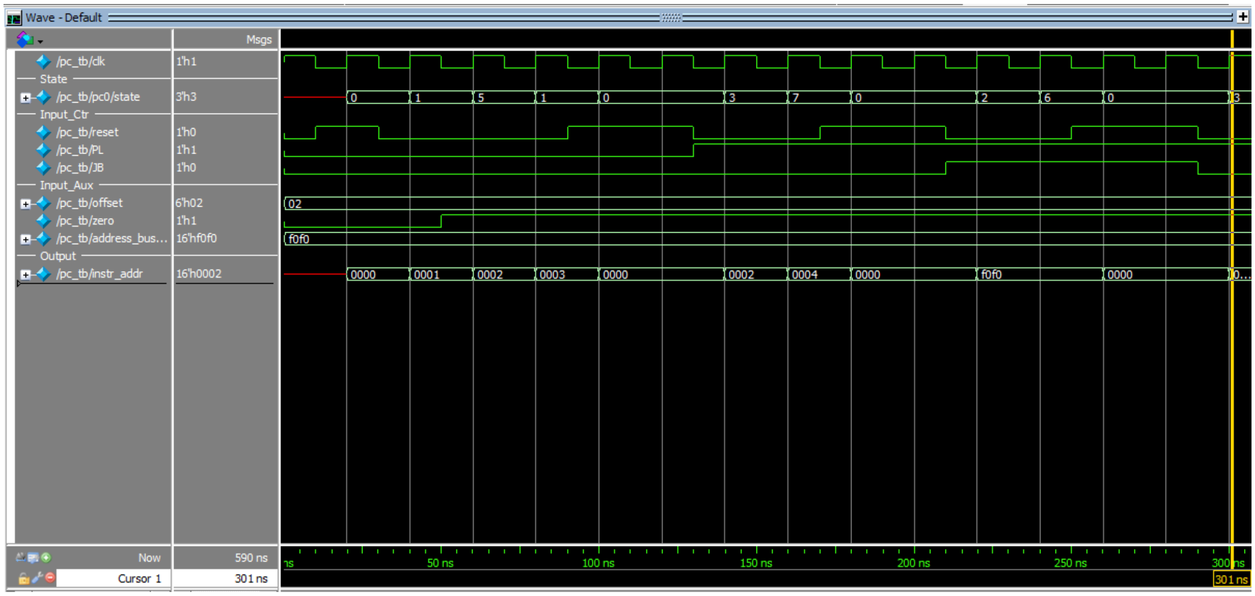


Figure 70 Simulation results for the Program Counter: part 1.

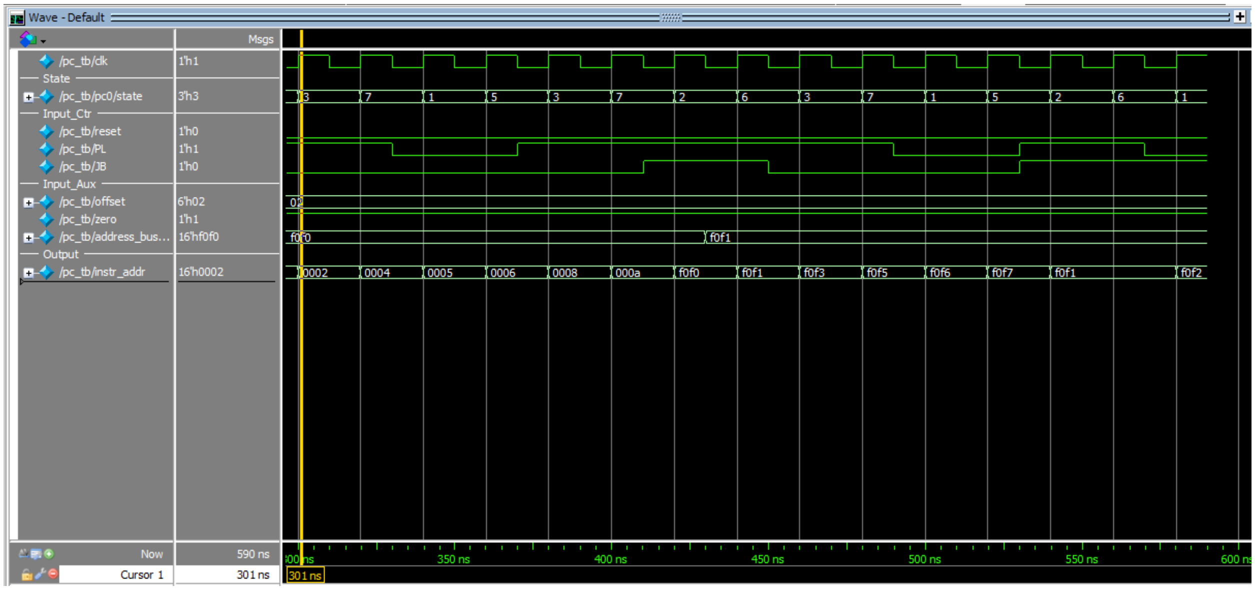


Figure 71 Simulation results for the Program Counter: part 2.

5.2.2.3 *Instruction Memory*

The Instruction Memory, described in section 3.1.2.2, contains a sample program. Providing sequential addresses as input leads to the matching instructions at the output.

Result: passed.

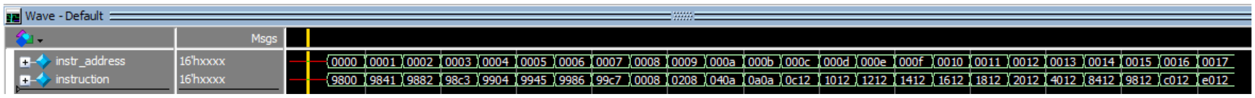
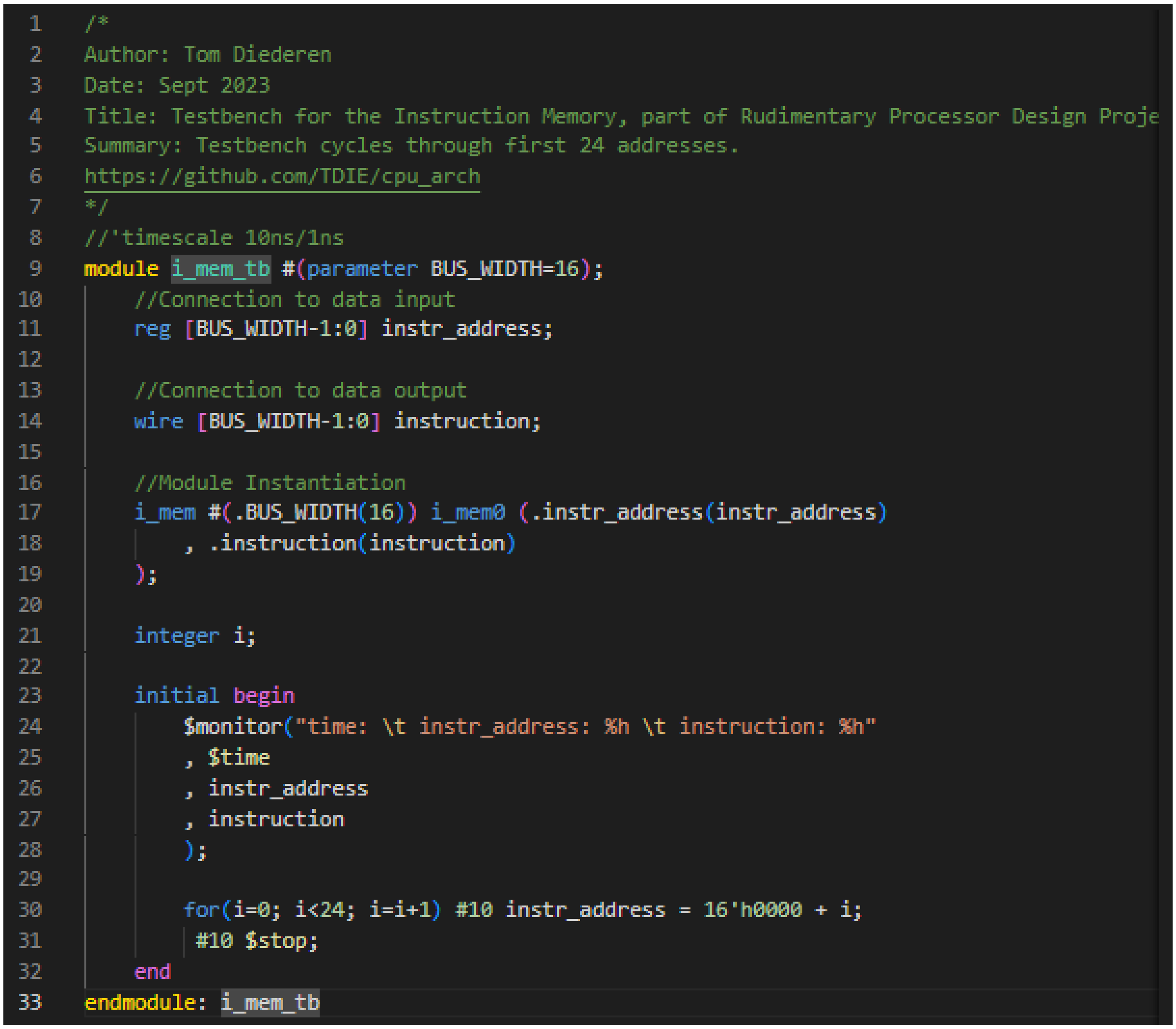

Wave - Default
Msgs
instr_address 16'hxxxx
0000 0001 0002 0003 0004 0005 0006 0007 0008 0009 000a 000b 000c 000d 000e 000f 0010 0011 0012 0013 0014 0015 0016 0017
instruction 16'hxxxx
9800 9841 9882 98c3 9904 9945 9986 99c7 0008 0208 040a 0a0a 0c12 1012 1212 1412 1612 1812 2012 4012 8412 9812 c012 e012

Figure 72 Simulation results for the Instruction Memory.



```
/*
Author: Tom Diederen
Date: Sept 2023
Title: Testbench for the Instruction Memory, part of Rudimentary Processor Design Proje
Summary: Testbench cycles through first 24 addresses.
https://github.com/TDIE/cpu_arch
*/
//'timescale 10ns/1ns
module i_mem_tb #(parameter BUS_WIDTH=16);
    //Connection to data input
    reg [BUS_WIDTH-1:0] instr_address;

    //Connection to data output
    wire [BUS_WIDTH-1:0] instruction;

    //Module Instantiation
    i_mem #(.BUS_WIDTH(16)) i_mem0 (.instr_address(instr_address)
        , .instruction(instruction)
    );

    integer i;

    initial begin
        $monitor("time: \t instr_address: %h \t instruction: %h"
        , $time
        , instr_address
        , instruction
        );

        for(i=0; i<24; i=i+1) #10 instr_address = 16'h0000 + i;
         #10 $stop;
    end
endmodule: i_mem_tb
```

Figure 73 Verilog testbench for the Instruction Memory.

5.2.2.4 Instruction Decoder

Test 1: Decode Each Major Instruction Type.

The instruction decoder, as described in section 3.1.2.3 on page 25, was tested by having it decode each major instruction type:

1. Using the Execution Unit and register(s)
2. Memory Read
3. Memory Write
4. EU with constant
5. Branch on Zero
6. Jump

Result: passed.

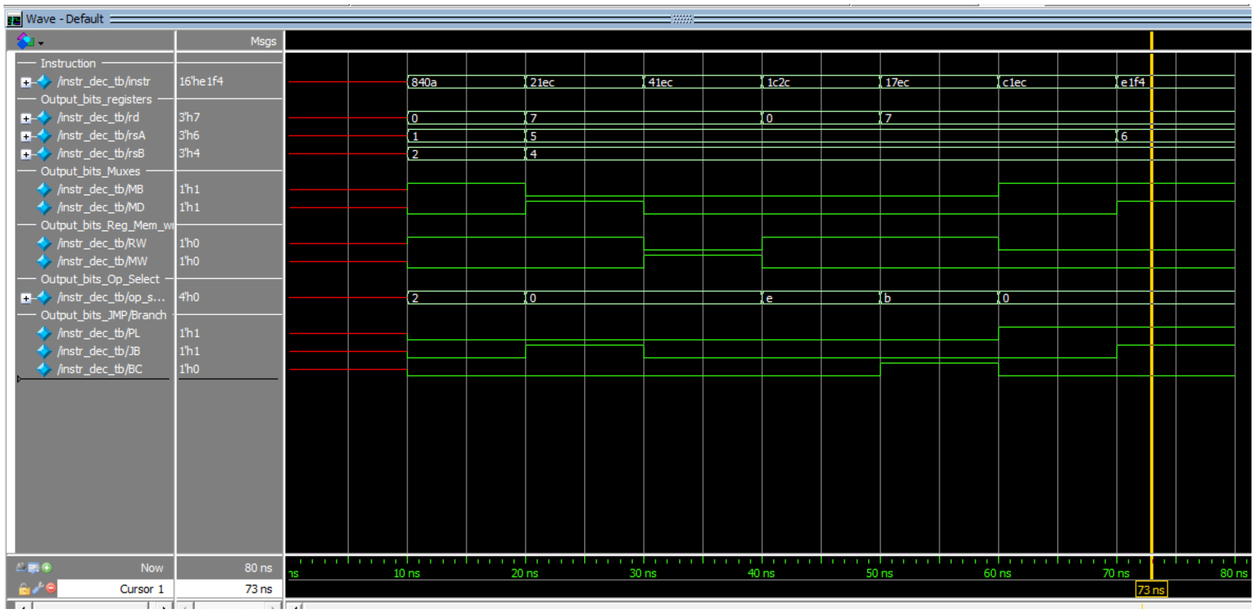
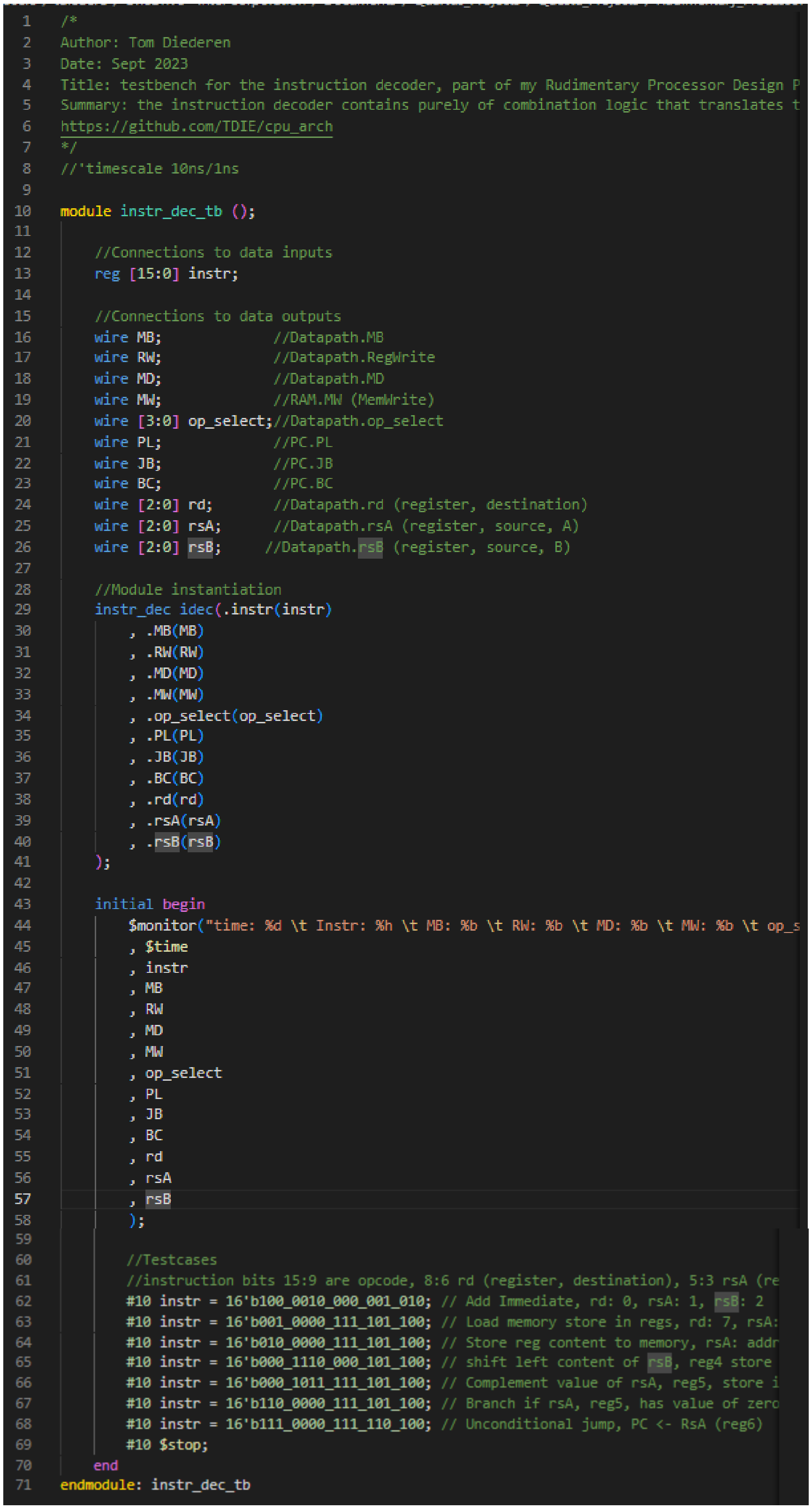


Figure 74 Simulation results for the Instruction Decoder.



```
/*
Author: Tom Diederen
Date: Sept 2023
Title: testbench for the instruction decoder, part of my Rudimentary Processor Design P
Summary: the instruction decoder contains purely of combination logic that translates t
https://github.com/TDIE/cpu_arch
*/
//'timescale 10ns/1ns

module instr_dec_tb ();

    //Connections to data inputs
    reg [15:0] instr;

    //Connections to data outputs
    wire MB;             //Datapath.MB
    wire RW;             //Datapath.RegWrite
    wire MD;             //Datapath.MD
    wire MW;             //RAM.MW (MemWrite)
    wire [3:0] op_select;//Datapath.op_select
    wire PL;             //PC.PL
    wire JB;             //PC.JB
    wire BC;             //PC.BC
    wire [2:0] rd;       //Datapath.rd (register, destination)
    wire [2:0] rsA;      //Datapath.rsA (register, source, A)
    wire [2:0] rsB;     //Datapath.rsB (register, source, B)

    //Module instantiation
    instr_dec idec(.instr(instr)
        , .MB(MB)
        , .RW(RW)
        , .MD(MD)
        , .MW(MW)
        , .op_select(op_select)
        , .PL(PL)
        , .JB(JB)
        , .BC(BC)
        , .rd(rd)
        , .rsA(rsA)
        , .rsB(rsB)
    );

    initial begin
        $monitor("time: %d \t Instr: %h \t MB: %b \t RW: %b \t MD: %b \t MW: %b \t op_s
        , $time
        , instr
        , MB
        , RW
        , MD
        , MW
        , op_select
        , PL
        , JB
        , BC
        , rd
        , rsA
        , rsB
        );

        //Testcases
        //instruction bits 15:9 are opcode, 8:6 rd (register, destination), 5:3 rsA (re
        #10 instr = 16'b100_0010_000_001_010; // Add Immediate, rd: 0, rsA: 1, rsB: 2
        #10 instr = 16'b001_0000_111_101_100; // Load memory store in regs, rd: 7, rsA:
        #10 instr = 16'b010_0000_111_101_100; // Store reg content to memory, rsA: addr
        #10 instr = 16'b000_1110_000_101_100; // shift left content of rsB, reg4 store
        #10 instr = 16'b000_1011_111_101_100; // Complement value of rsA, reg5, store i
        #10 instr = 16'b110_0000_111_101_100; // Branch if rsA, reg5, has value of zero
        #10 instr = 16'b111_0000_111_110_100; // Unconditional jump, PC <- RsA (reg6)
        #10 $stop;
    end
endmodule: instr_dec_tb
```

Figure 75 Testbench for the Instruction Decoder

5.3 System Level Testbench (UVM)

Custom made testbenches, as described in the previous section, scale poorly and provide minimal re-use opportunity. There's also a limit to manually analyzing the waveforms. In order to provide an automated, scalable, and re-usable verification environment, the Universal Verification Methodology was used for testing the processor at the system level.

Creating a quick testbench in Verilog for the system level is relatively straightforward, however analyzing it by hand would be quite time consuming. The limitations of the verification method used in the previous section become evident:

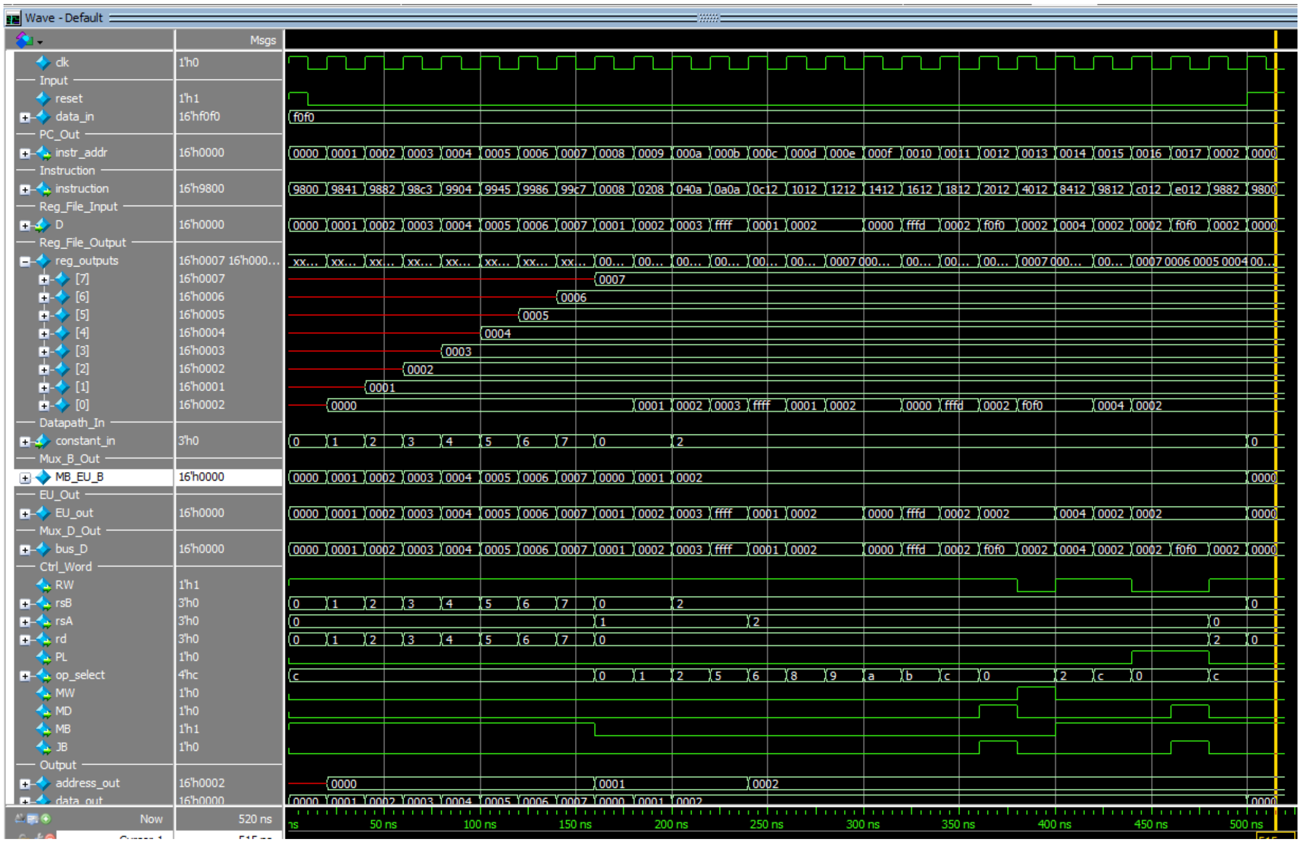


Figure 76 System Level Testbench for the CPU.

5.3.1 Verification Strategy

Since this was an after-hours, one-person, project, the scope had to be kept manageable. The design's instruction memory contains a test program that performs each instruction once, while using all registers. The system level verification goal is to make sure that each instruction yields the correct result. Individual components and their functionality, e.g. read from and write to registers, have already been verified in the previous section.

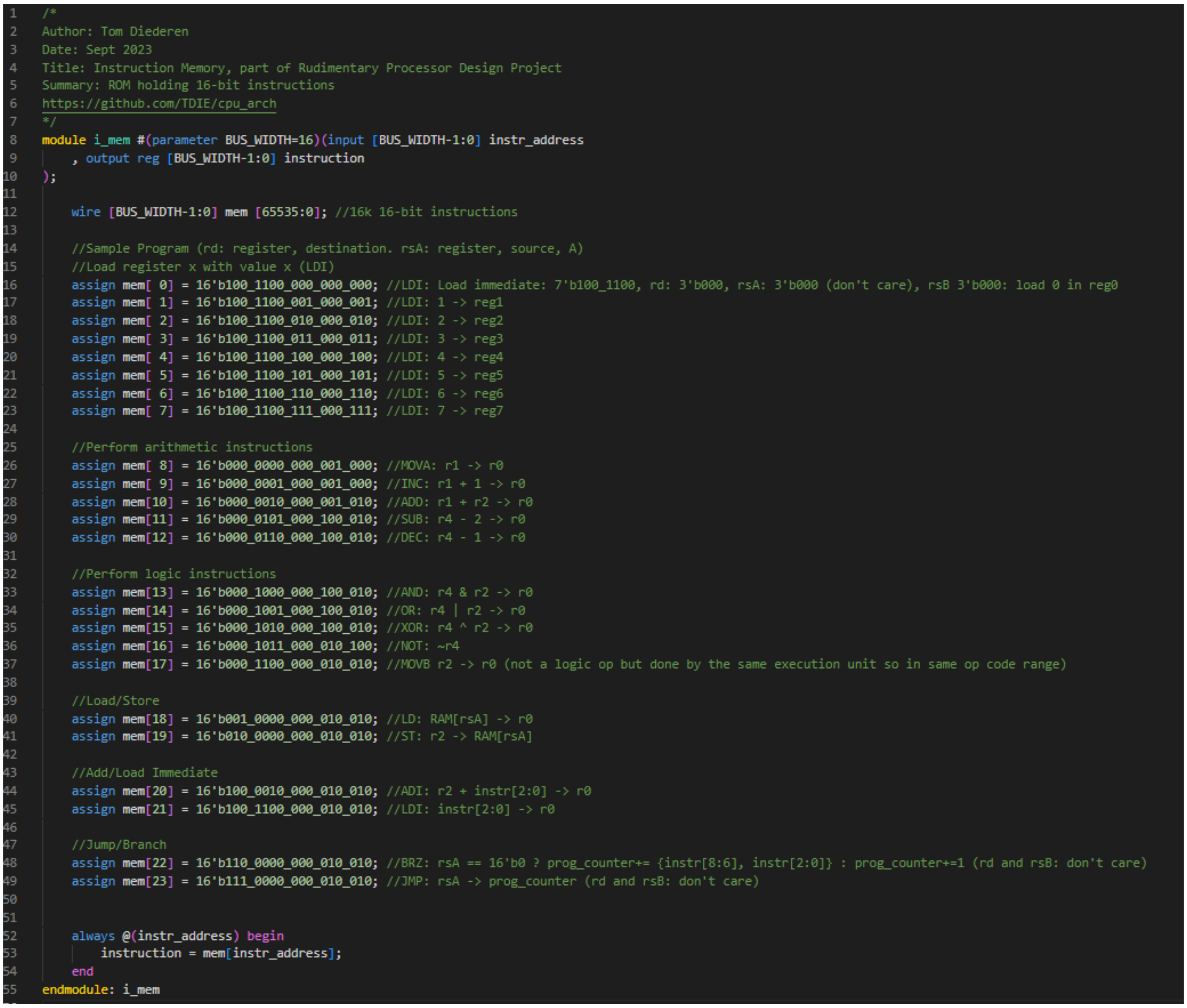
5.3.1.1 Constraints

The scope of the UVM testbench has been limited in the following ways:

- The instruction register is a ROM. Only a pre-written program in the instruction memory will be verified. The program executes each type of instruction and uses all registers.

- Because the instruction register is read-only, no register abstraction layer will be created.
- Verification of Read and Write operations from/to external RAM are out of scope. The testbench will simulate input data from RAM and no write backs will be tested.
- Communication with the CPU will happen through a single signal interface, called the cpu interface, which does not adhere to any official bus standard.

The branch on zero and jump instructions will not be verified at the system level. They require a little more complexity on the monitor side to capture the current PC value, as well as the next one. Thes BRZ and JMP instructions have been verified on the sub-block level in section



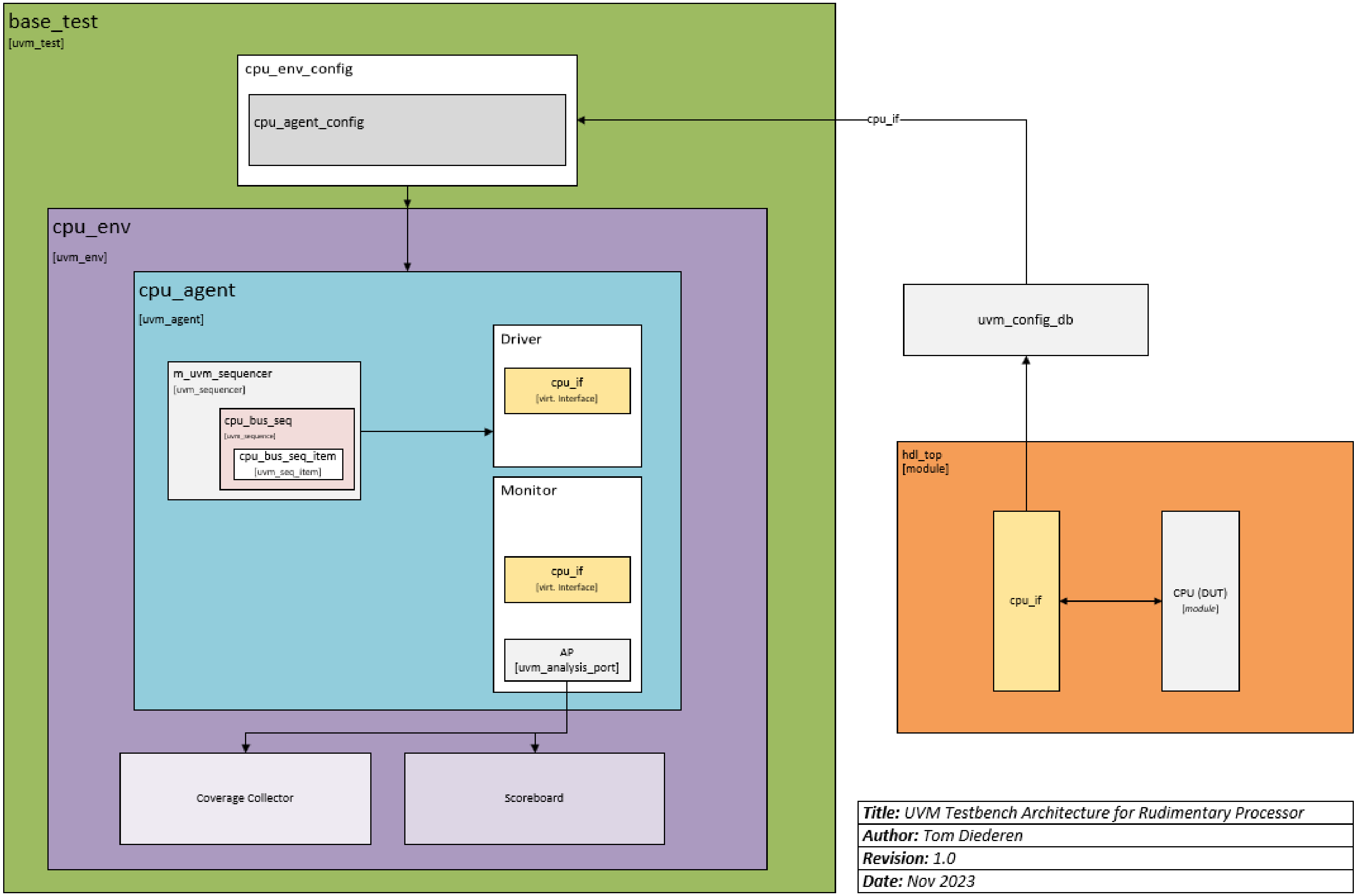
```
/*
Author: Tom Diederen
Date: Sept 2023
Title: Instruction Memory, part of Rudimentary Processor Design Project
Summary: ROM holding 16-bit instructions
https://github.com/TDIE/cpu_arch
*/
module i_mem #(parameter BUS_WIDTH=16)(input [BUS_WIDTH-1:0] instr_address
    , output reg [BUS_WIDTH-1:0] instruction
);

    wire [BUS_WIDTH-1:0] mem [65535:0]; //16k 16-bit instructions

    //Sample Program (rd: register, destination. rsA: register, source, A)
    //Load register x with value x (LDI)
    assign mem[ 0] = 16'b100_1100_000_000_000; //LDI: Load immediate: 7'b100_1100, rd: 3'b000, rsA: 3'b000 (don't care), rsB 3'b000: load 0 in reg0
    assign mem[ 1] = 16'b100_1100_001_000_001; //LDI: 1 -> reg1
    assign mem[ 2] = 16'b100_1100_010_000_010; //LDI: 2 -> reg2
    assign mem[ 3] = 16'b100_1100_011_000_011; //LDI: 3 -> reg3
    assign mem[ 4] = 16'b100_1100_100_000_100; //LDI: 4 -> reg4
    assign mem[ 5] = 16'b100_1100_101_000_101; //LDI: 5 -> reg5
    assign mem[ 6] = 16'b100_1100_110_000_110; //LDI: 6 -> reg6
    assign mem[ 7] = 16'b100_1100_111_000_111; //LDI: 7 -> reg7

    //Perform arithmetic instructions
    assign mem[ 8] = 16'b000_0000_000_001_000; //MOVA: r1 -> r0
    assign mem[ 9] = 16'b000_0001_000_001_000; //INC: r1 + 1 -> r0
    assign mem[10] = 16'b000_0010_000_001_010; //ADD: r1 + r2 -> r0
    assign mem[11] = 16'b000_0101_000_100_010; //SUB: r4 - 2 -> r0
    assign mem[12] = 16'b000_0110_000_100_010; //DEC: r4 - 1 -> r0

    //Perform logic instructions
    assign mem[13] = 16'b000_1000_000_100_010; //AND: r4 & r2 -> r0
    assign mem[14] = 16'b000_1001_000_100_010; //OR: r4 | r2 -> r0
    assign mem[15] = 16'b000_1010_000_100_010; //XOR: r4 ^ r2 -> r0
    assign mem[16] = 16'b000_1011_000_010_100; //NOT: ~r4
    assign mem[17] = 16'b000_1100_000_010_010; //MOVB r2 -> r0 (not a logic op but done by the same execution unit so in same op code range)

    //Load/Store
    assign mem[18] = 16'b001_0000_000_010_010; //LD: RAM[rsA] -> r0
    assign mem[19] = 16'b010_0000_000_010_010; //ST: r2 -> RAM[rsA]

    //Add/Load Immediate
    assign mem[20] = 16'b100_0010_000_010_010; //ADI: r2 + instr[2:0] -> r0
    assign mem[21] = 16'b100_1100_000_010_010; //LDI: instr[2:0] -> r0

    //Jump/Branch
    assign mem[22] = 16'b110_0000_000_010_010; //BRZ: rsA == 16'b0 ? prog_counter+= {instr[8:6], instr[2:0]} : prog_counter+=1 (rd and rsB: don't care)
    assign mem[23] = 16'b111_0000_000_010_010; //JMP: rsA -> prog_counter (rd and rsB: don't care)


    always @(instr_address) begin
        instruction = mem[instr_address];
    end
endmodule: i_mem
```

Figure 68 The sample program in the instruction memory.

- Program Counter0 on page 67.

5.3.2 Testbench Architecture

The testbench will be built using a block level architecture as shown in Figure 77. The CPU is connected to the testbench through an interface: the cpu interface. A handle to it is stored in the UVM config database. The testbench only uses one test: the base test. This test configures the environment for the cpu interface as well as its agent according to configuration objects. The agent contains the monitor, driver, and sequencer. The environment contains a scoreboard and coverage collector (although the latter is unsupported in the free Questa version that was used for this project). The next sections describe each part of the testbench in more detail.



Title: *UVM Testbench Architecture for Rudimentary Processor*
Author: *Tom Diederen*
Revision: *1.0*
Date: *Nov 2023*

Figure 77 High-Level Architecture of the UVM Testbench

5.3.2.1 Interface

The testbench contains a single interface to communicate with the DUT: the cpu_if. It provides the clock, reset, and external data input, which simulates output from RAM to the DUT. It is custom made for this project and doesn't implement any official protocol.

The signals sent from the interface to the testbench will be used to predict, and verify, which instruction was executed and whether the result is correct. The interface is bi-directional and non-pipelined. A schematic diagram, signal overview, and the SystemVerilog code are shown below.

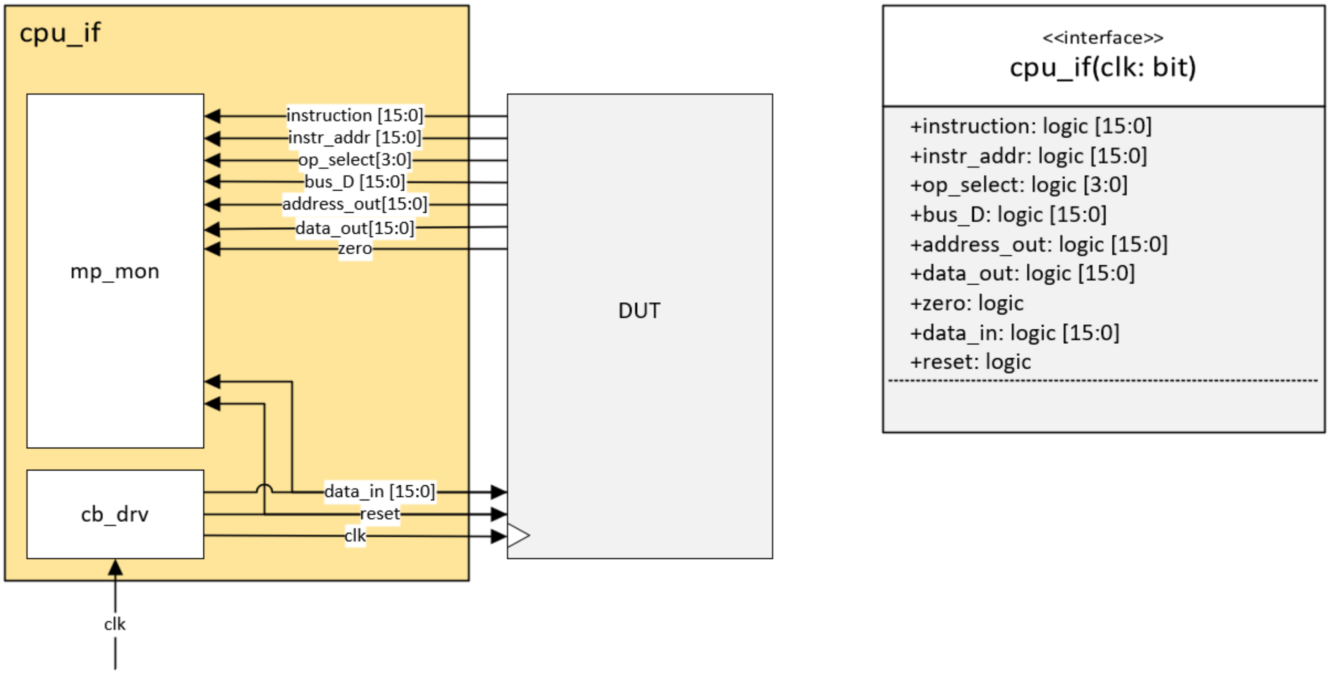
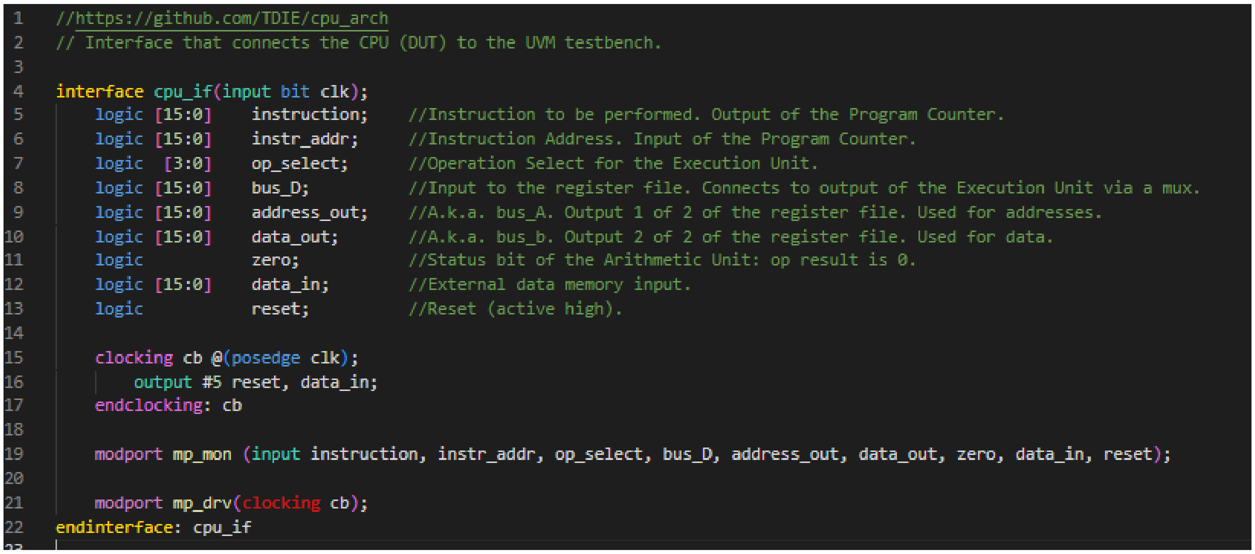


Figure 78 Overview and UML Diagram of the Interface.

Table 8 Interface Signal Overview

Signal	Description	Origin
clk	Clock signal	Top lvl input
reset	Reset	Top lvl input
data_in [15:0]	Data input from RAM	Top lvl Input
Instruction [15:0]	Instruction to be executed	Ctrl_top.i_mem
instr_addr [15:0]	Address for instr. register	Ctrl_top.pc
bus_D [15:0]	Muxed: Execution Unit output or input from memory (data_in[15:0])	Dp_top
address_out [15:0]	Address to PC / Memory (Bus A)	Dp_top
data_out [15:0]	Data to Memory (Bus B)	Dp_top
zero	ALU Status Bit	Dp_top.eu.arithmetic
op_select [3:0]	Selects execution unit operation to be performed	Ctrl_top.instr_dec



```
//https://github.com/TDIE/cpu_arch
// Interface that connects the CPU (DUT) to the UVM testbench.

interface cpu_if(input bit clk);
    logic [15:0]    instruction;    //Instruction to be performed. Output of the Program Counter.
    logic [15:0]    instr_addr;     //Instruction Address. Input of the Program Counter.
    logic  [3:0]    op_select;      //Operation Select for the Execution Unit.
    logic [15:0]    bus_D;          //Input to the register file. Connects to output of the Execution Unit via a mux.
    logic [15:0]    address_out;    //A.k.a. bus_A. Output 1 of 2 of the register file. Used for addresses.
    logic [15:0]    data_out;       //A.k.a. bus_b. Output 2 of 2 of the register file. Used for data.
    logic           zero;           //Status bit of the Arithmetic Unit: op result is 0.
    logic [15:0]    data_in;        //External data memory input.
    logic           reset;          //Reset (active high).

    clocking cb @(posedge clk);
        output #5 reset, data_in;
    endclocking: cb

    modport mp_mon (input instruction, instr_addr, op_select, bus_D, address_out, data_out, zero, data_in, reset);

    modport mp_drv(clocking cb);
endinterface: cpu_if
```

Figure 79 Design of the CPU interface.

5.3.2.2 Test

The testbench uses a single test which creates the environment based on a configuration object. An overview and UML diagrams are shown below.

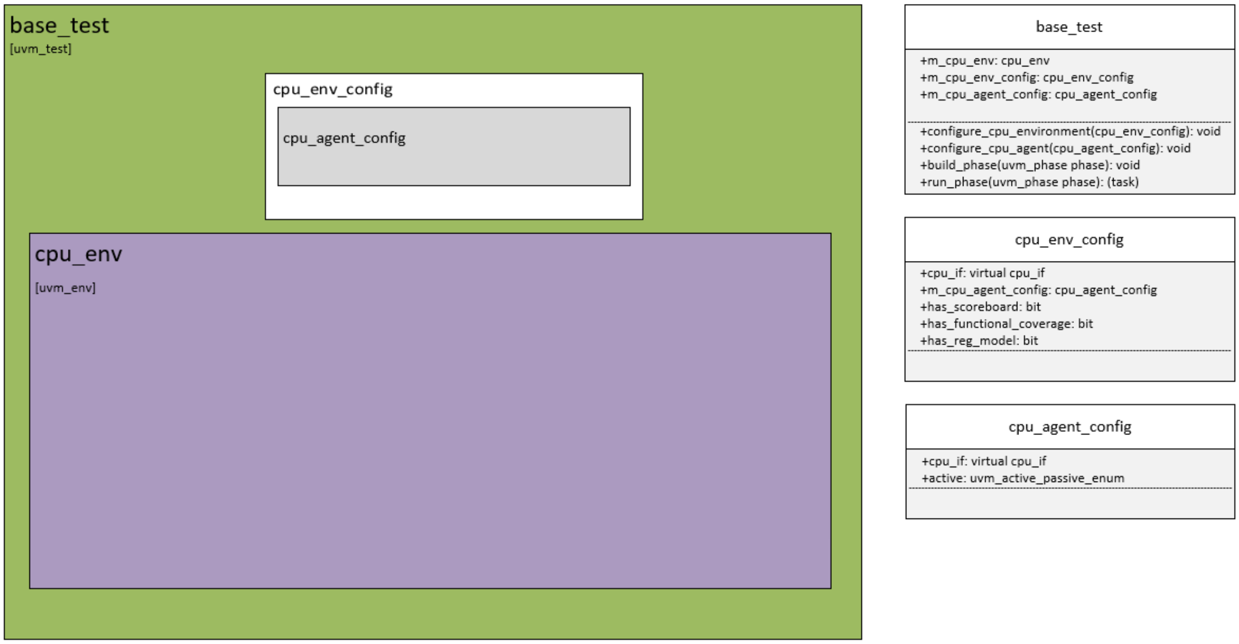
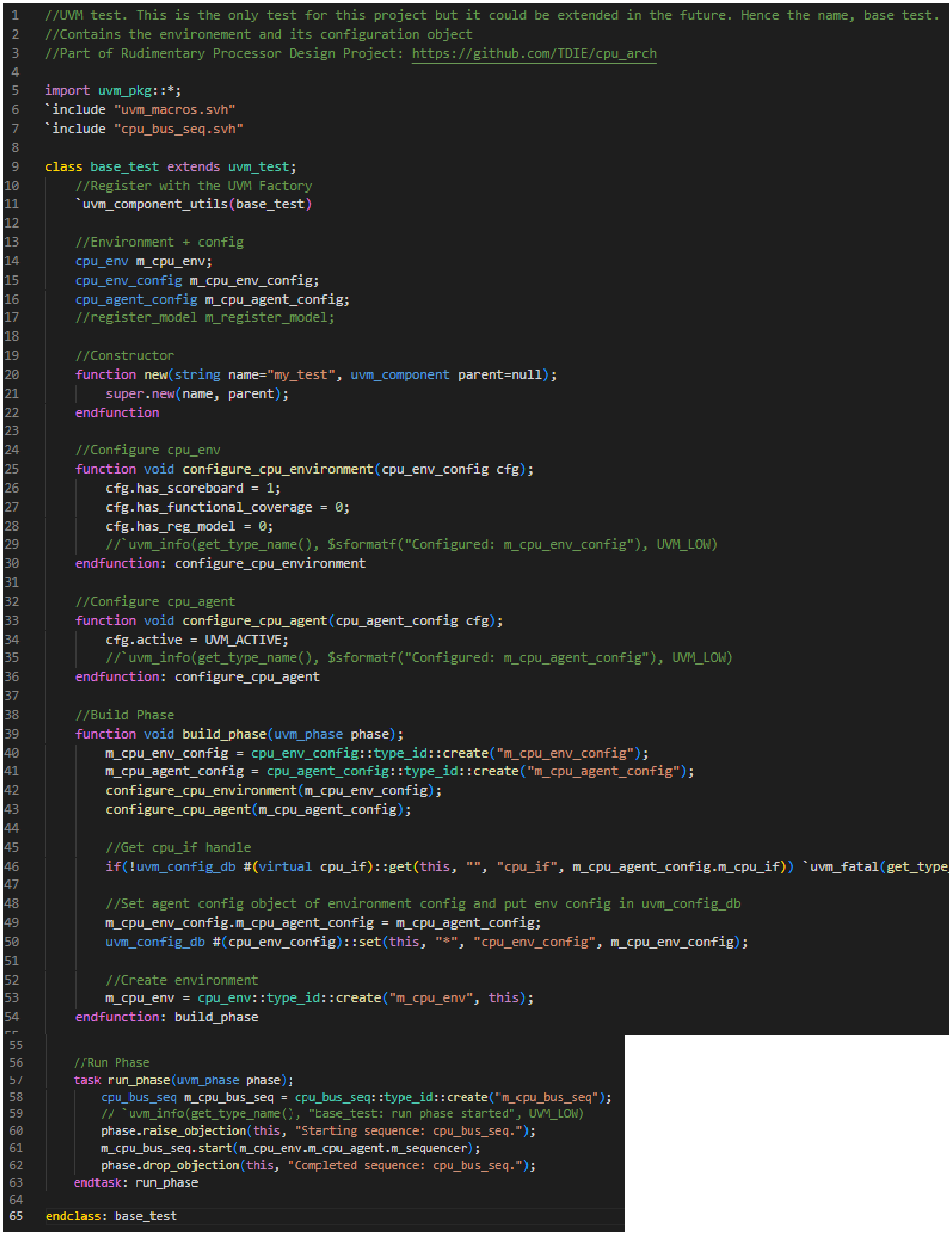
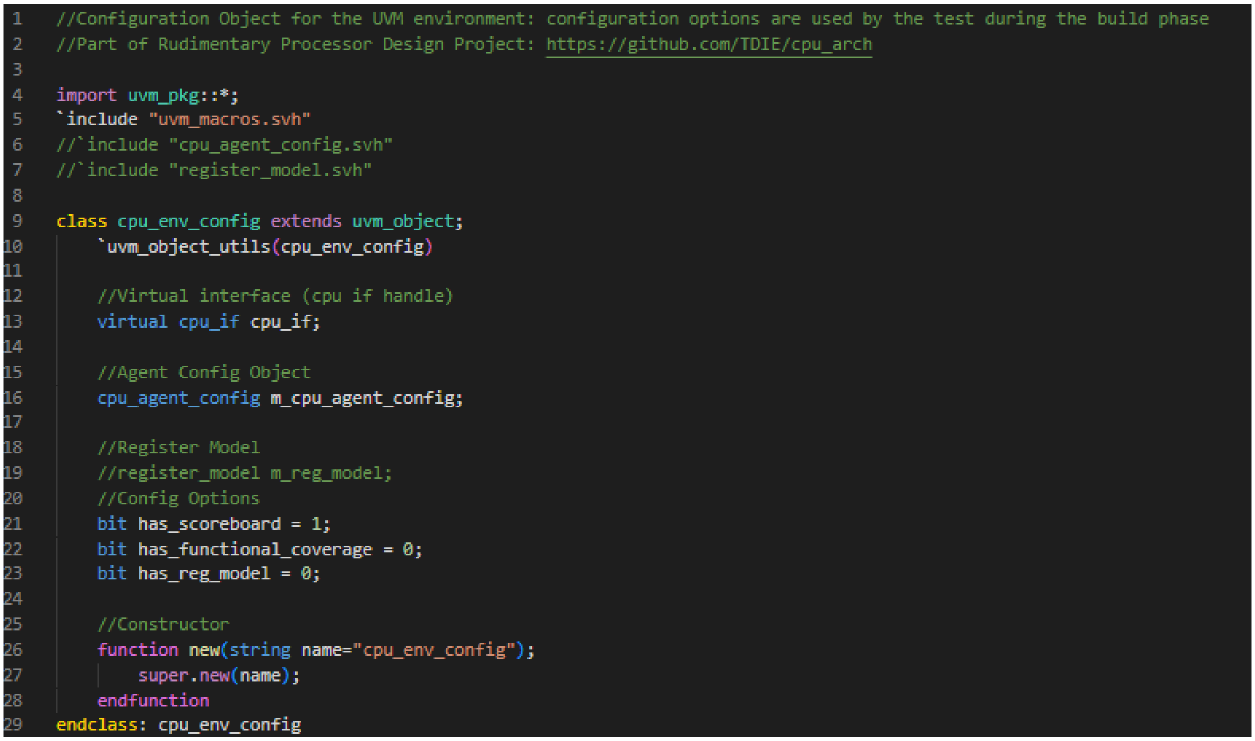

base_test
[uvm_test]
cpu_env_config
cpu_agent_config
cpu_env
[uvm_env]
base_test
+m_cpu_env: cpu_env
+m_cpu_env_config: cpu_env_config
+m_cpu_agent_config: cpu_agent_config
+configure_cpu_environment(cpu_env_config): void
+configure_cpu_agent(cpu_agent_config): void
+build_phase(uvm_phase phase): void
+run_phase(uvm_phase phase): (task)
cpu_env_config
+cpu_if: virtual cpu_if
+m_cpu_agent_config: cpu_agent_config
+has_scoreboard: bit
+has_functional_coverage: bit
+has_reg_model: bit
cpu_agent_config
+cpu_if: virtual cpu_if
+active: uvm_active_passive_enum

Figure 80 Overview and UML Diagram of the UVM Test.



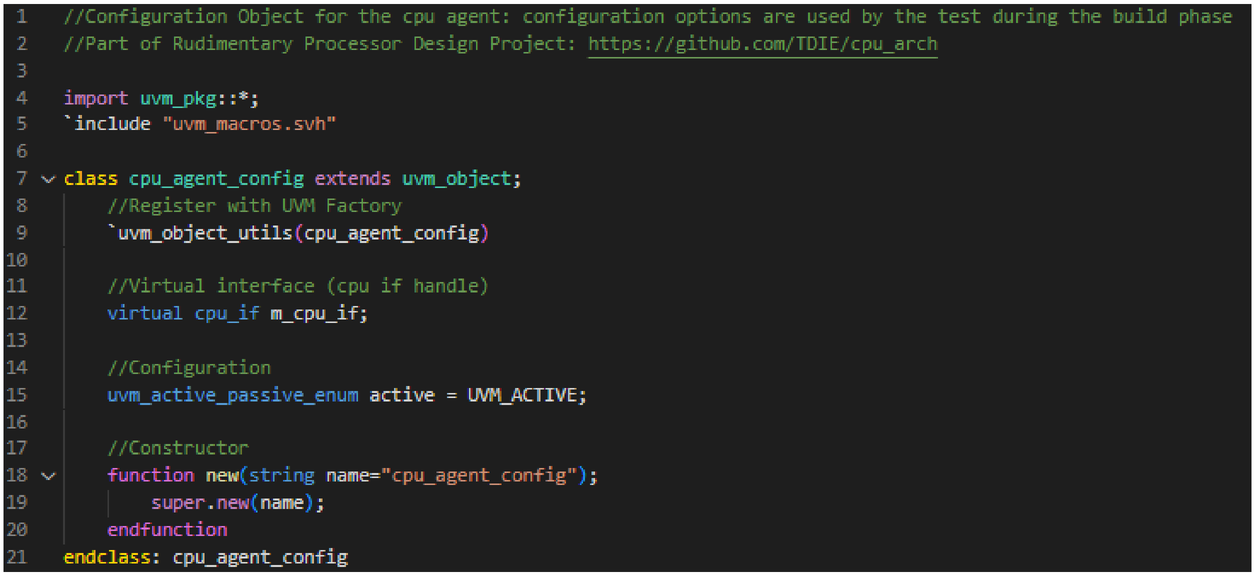
```
//UVM test. This is the only test for this project but it could be extended in the future. Hence the name, base test.
//Contains the environement and its configuration object
//Part of Rudimentary Processor Design Project: https://github.com/TDIE/cpu_arch

import uvm_pkg::*;
`include "uvm_macros.svh"
`include "cpu_bus_seq.svh"

class base_test extends uvm_test;
    //Register with the UVM Factory
    `uvm_component_utils(base_test)

    //Environment + config
    cpu_env m_cpu_env;
    cpu_env_config m_cpu_env_config;
    cpu_agent_config m_cpu_agent_config;
    //register_model m_register_model;

    //Constructor
    function new(string name="my_test", uvm_component parent=null);
        super.new(name, parent);
    endfunction

    //Configure cpu_env
    function void configure_cpu_environment(cpu_env_config cfg);
        cfg.has_scoreboard = 1;
        cfg.has_functional_coverage = 0;
        cfg.has_reg_model = 0;
        //`uvm_info(get_type_name(), $sformatf("Configured: m_cpu_env_config"), UVM_LOW)
    endfunction: configure_cpu_environment

    //Configure cpu_agent
    function void configure_cpu_agent(cpu_agent_config cfg);
        cfg.active = UVM_ACTIVE;
        //`uvm_info(get_type_name(), $sformatf("Configured: m_cpu_agent_config"), UVM_LOW)
    endfunction: configure_cpu_agent

    //Build Phase
    function void build_phase(uvm_phase phase);
        m_cpu_env_config = cpu_env_config::type_id::create("m_cpu_env_config");
        m_cpu_agent_config = cpu_agent_config::type_id::create("m_cpu_agent_config");
        configure_cpu_environment(m_cpu_env_config);
        configure_cpu_agent(m_cpu_agent_config);

        //Get cpu_if handle
        if(!uvm_config_db #(virtual cpu_if)::get(this, "", "cpu_if", m_cpu_agent_config.m_cpu_if)) `uvm_fatal(get_type

        //Set agent config object of environment config and put env config in uvm_config_db
        m_cpu_env_config.m_cpu_agent_config = m_cpu_agent_config;
        uvm_config_db #(cpu_env_config)::set(this, "*", "cpu_env_config", m_cpu_env_config);

        //Create environment
        m_cpu_env = cpu_env::type_id::create("m_cpu_env", this);
    endfunction: build_phase

    //Run Phase
    task run_phase(uvm_phase phase);
        cpu_bus_seq m_cpu_bus_seq = cpu_bus_seq::type_id::create("m_cpu_bus_seq");
        // `uvm_info(get_type_name(), "base_test: run phase started", UVM_LOW)
        phase.raise_objection(this, "Starting sequence: cpu_bus_seq.");
        m_cpu_bus_seq.start(m_cpu_env.m_cpu_agent.m_sequencer);
        phase.drop_objection(this, "Completed sequence: cpu_bus_seq.");
    endtask: run_phase

endclass: base_test
```

Figure 81 Source code of the UVM base test.



```
//Configuration Object for the UVM environment: configuration options are used by the test during the build phase
//Part of Rudimentary Processor Design Project: https://github.com/TDIE/cpu_arch

import uvm_pkg::*;
`include "uvm_macros.svh"
//`include "cpu_agent_config.svh"
//`include "register_model.svh"

class cpu_env_config extends uvm_object;
    `uvm_object_utils(cpu_env_config)

    //Virtual interface (cpu if handle)
    virtual cpu_if cpu_if;

    //Agent Config Object
    cpu_agent_config m_cpu_agent_config;

    //Register Model
    //register_model m_reg_model;
    //Config Options
    bit has_scoreboard = 1;
    bit has_functional_coverage = 0;
    bit has_reg_model = 0;

    //Constructor
    function new(string name="cpu_env_config");
        super.new(name);
    endfunction
endclass: cpu_env_config
```

Figure 82 Source code of the CPU Environment Configuration Class (UVM).



```
//Configuration Object for the cpu agent: configuration options are used by the test during the build phase
//Part of Rudimentary Processor Design Project: https://github.com/TDIE/cpu_arch

import uvm_pkg::*;
`include "uvm_macros.svh"

class cpu_agent_config extends uvm_object;
    //Register with UVM Factory
    `uvm_object_utils(cpu_agent_config)

    //Virtual interface (cpu if handle)
    virtual cpu_if m_cpu_if;

    //Configuration
    uvm_active_passive_enum active = UVM_ACTIVE;

    //Constructor
    function new(string name="cpu_agent_config");
        super.new(name);
    endfunction
endclass: cpu_agent_config
```

Figure 83 Source code of the CPU Agent Configuration Class (UVM).

5.3.2.3 Environment

The environment contains the agent, scoreboard, and coverage collector. The CPU has only one interface so a single agent will be created. An overview as well as the source code are shown below. Note that the source code for the config class was already shown in the previous section.

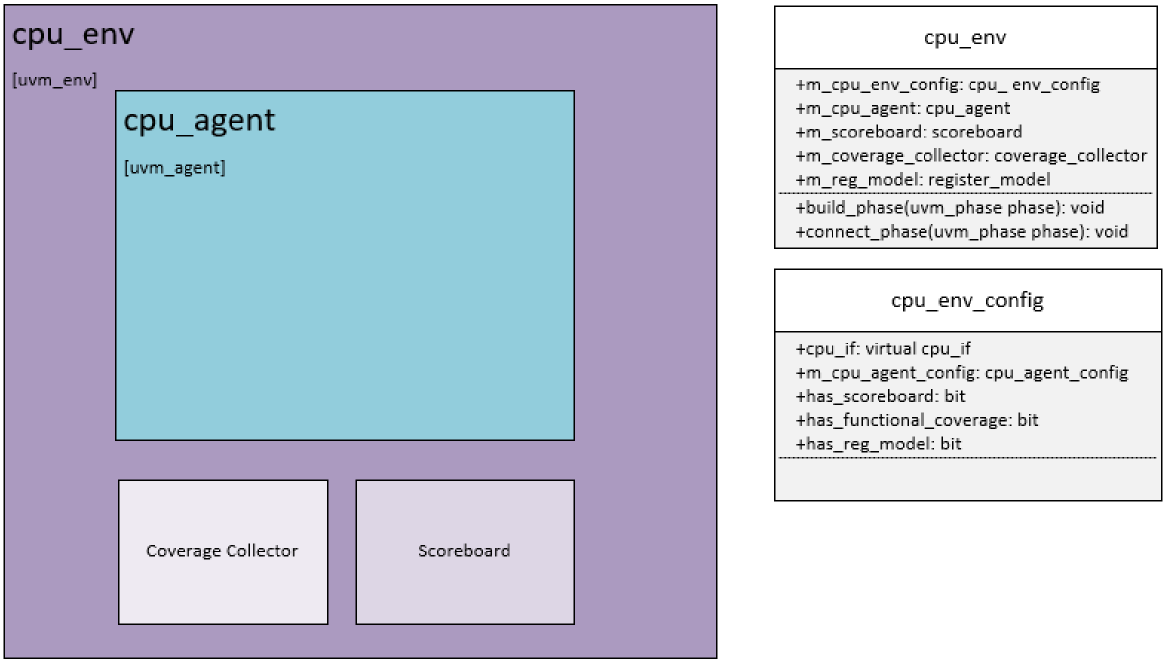
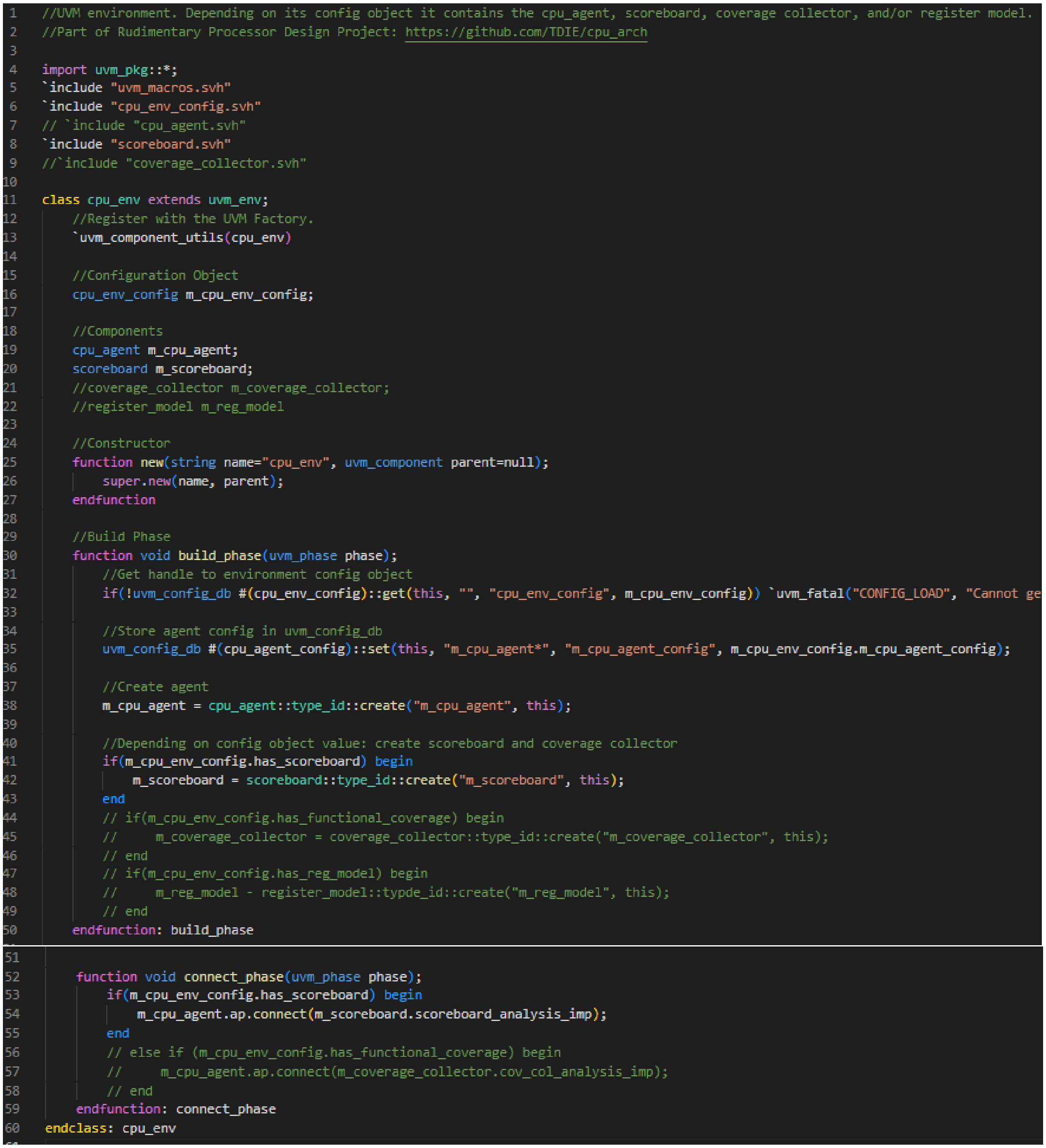

cpu_env
[uvm_env]
cpu_agent
[uvm_agent]
Coverage Collector
Scoreboard
cpu_env
+m_cpu_env_config: cpu_ env_config
+m_cpu_agent: cpu_agent
+m_scoreboard: scoreboard
+m_coverage_collector: coverage_collector
+m_reg_model: register_model
+build_phase(uvm_phase phase): void
+connect_phase(uvm_phase phase): void
cpu_env_config
+cpu_if: virtual cpu_if
+m_cpu_agent_config: cpu_agent_config
+has_scoreboard: bit
+has_functional_coverage: bit
+has_reg_model: bit

Figure 84 Overview and UML Diagram of the Environment.



```
//UVM environment. Depending on its config object it contains the cpu_agent, scoreboard, coverage collector, and/or register model.
//Part of Rudimentary Processor Design Project: https://github.com/TDIE/cpu_arch

import uvm_pkg::*;
`include "uvm_macros.svh"
`include "cpu_env_config.svh"
// `include "cpu_agent.svh"
`include "scoreboard.svh"
//`include "coverage_collector.svh"

class cpu_env extends uvm_env;
    //Register with the UVM Factory.
    `uvm_component_utils(cpu_env)

    //Configuration Object
    cpu_env_config m_cpu_env_config;

    //Components
    cpu_agent m_cpu_agent;
    scoreboard m_scoreboard;
    //coverage_collector m_coverage_collector;
    //register_model m_reg_model

    //Constructor
    function new(string name="cpu_env", uvm_component parent=null);
        super.new(name, parent);
    endfunction

    //Build Phase
    function void build_phase(uvm_phase phase);
        //Get handle to environment config object
        if(!uvm_config_db #(cpu_env_config)::get(this, "", "cpu_env_config", m_cpu_env_config)) `uvm_fatal("CONFIG_LOAD", "Cannot ge

        //Store agent config in uvm_config_db
        uvm_config_db #(cpu_agent_config)::set(this, "m_cpu_agent*", "m_cpu_agent_config", m_cpu_env_config.m_cpu_agent_config);

        //Create agent
        m_cpu_agent = cpu_agent::type_id::create("m_cpu_agent", this);

        //Depending on config object value: create scoreboard and coverage collector
        if(m_cpu_env_config.has_scoreboard) begin
            m_scoreboard = scoreboard::type_id::create("m_scoreboard", this);
        end
        // if(m_cpu_env_config.has_functional_coverage) begin
        //     m_coverage_collector = coverage_collector::type_id::create("m_coverage_collector", this);
        // end
        // if(m_cpu_env_config.has_reg_model) begin
        //     m_reg_model - register_model::typde_id::create("m_reg_model", this);
        // end
    endfunction: build_phase

    function void connect_phase(uvm_phase phase);
        if(m_cpu_env_config.has_scoreboard) begin
            m_cpu_agent.ap.connect(m_scoreboard.scoreboard_analysis_imp);
        end
        // else if (m_cpu_env_config.has_functional_coverage) begin
        //     m_cpu_agent.ap.connect(m_coverage_collector.cov_col_analysis_imp);
        // end
    endfunction: connect_phase
endclass: cpu_env
```

Figure 85 Source code of the CPU environment (UVM).

5.3.2.4 Agent

The agent contains a monitor, driver, and sequencer that runs the CPU bus sequence. The agent can be configured during the build phase through its configuration object. The configuration object also holds a handle to the virtual interface that the driver and monitor will use. An overview is shown below. Note that the source code of the config classes was already shown in the section on the UVM test: section 5.3.2.2 on page 79. The bus sequence and sequence items are described in section 5.3.2.5 on page 88.

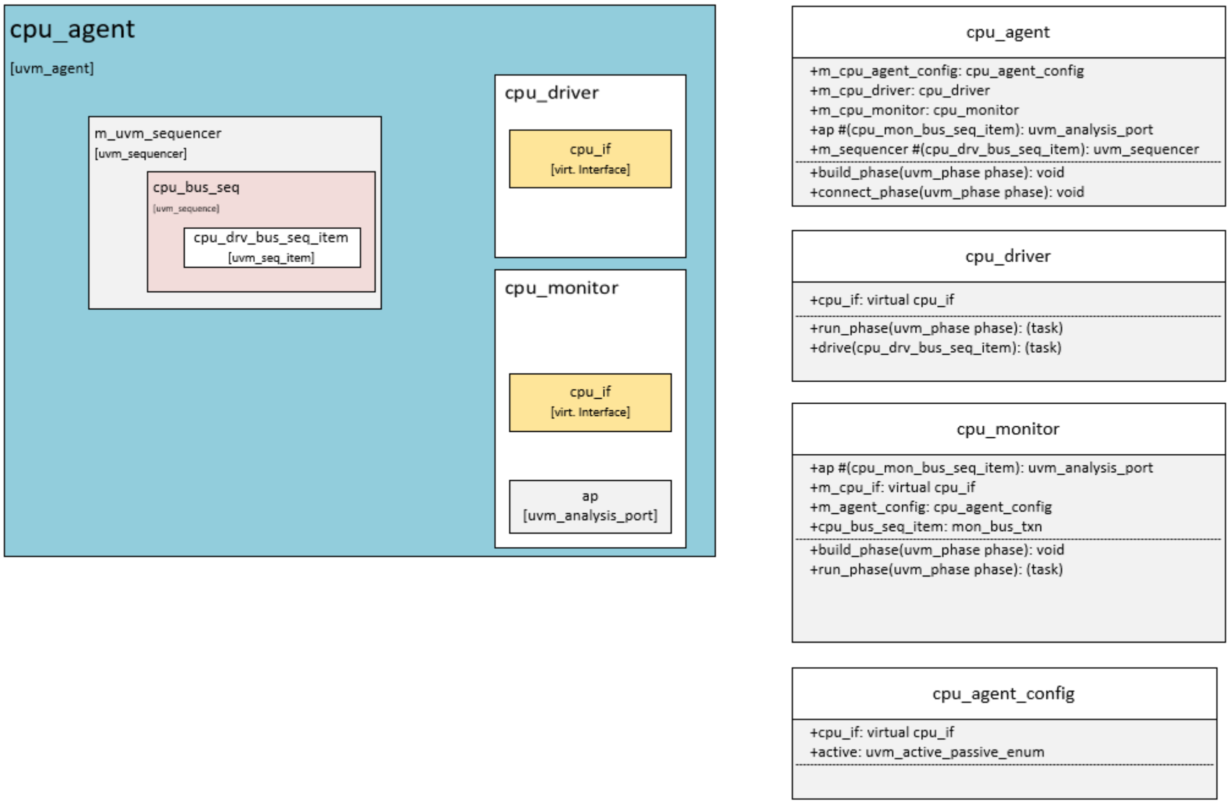
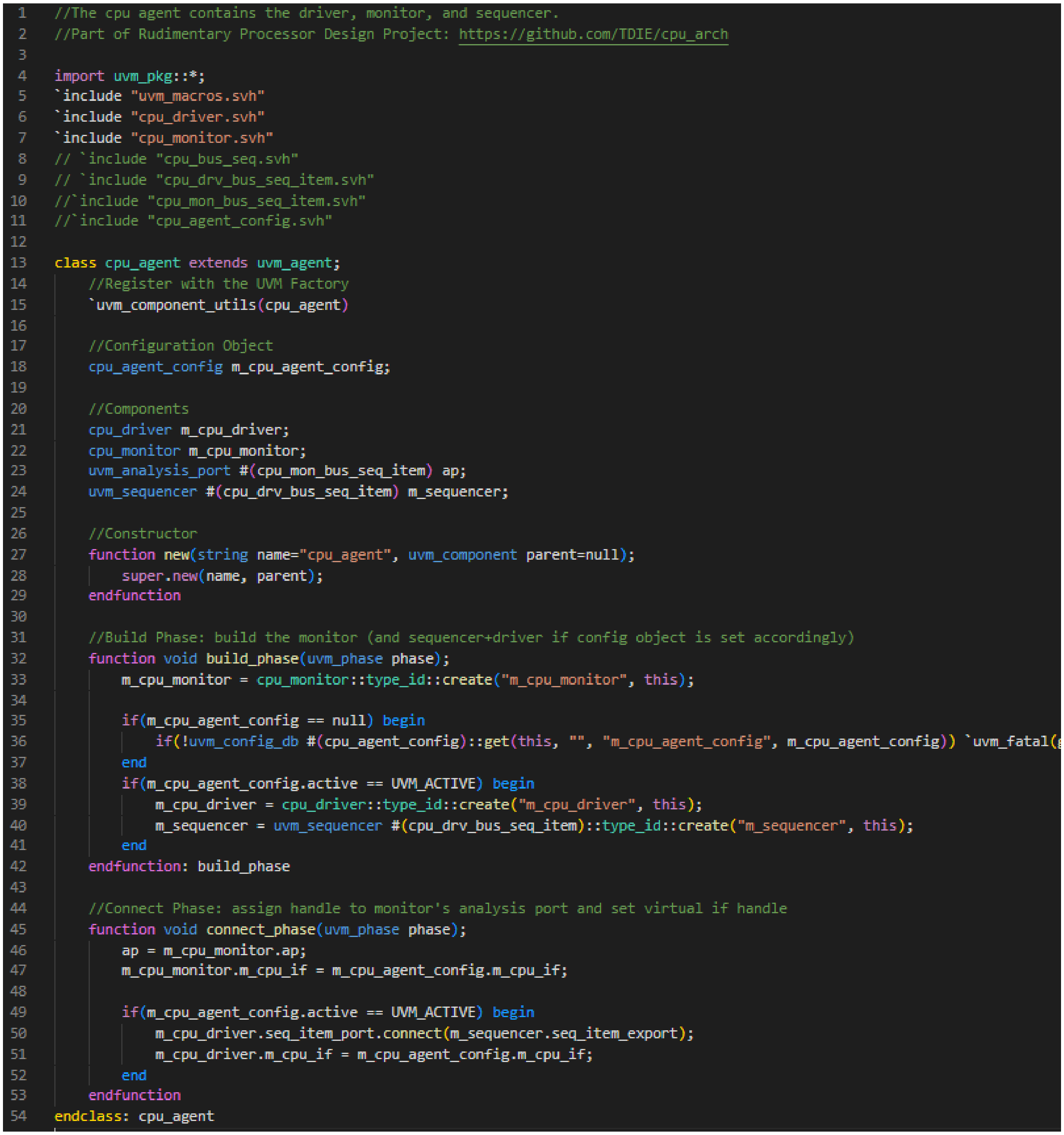
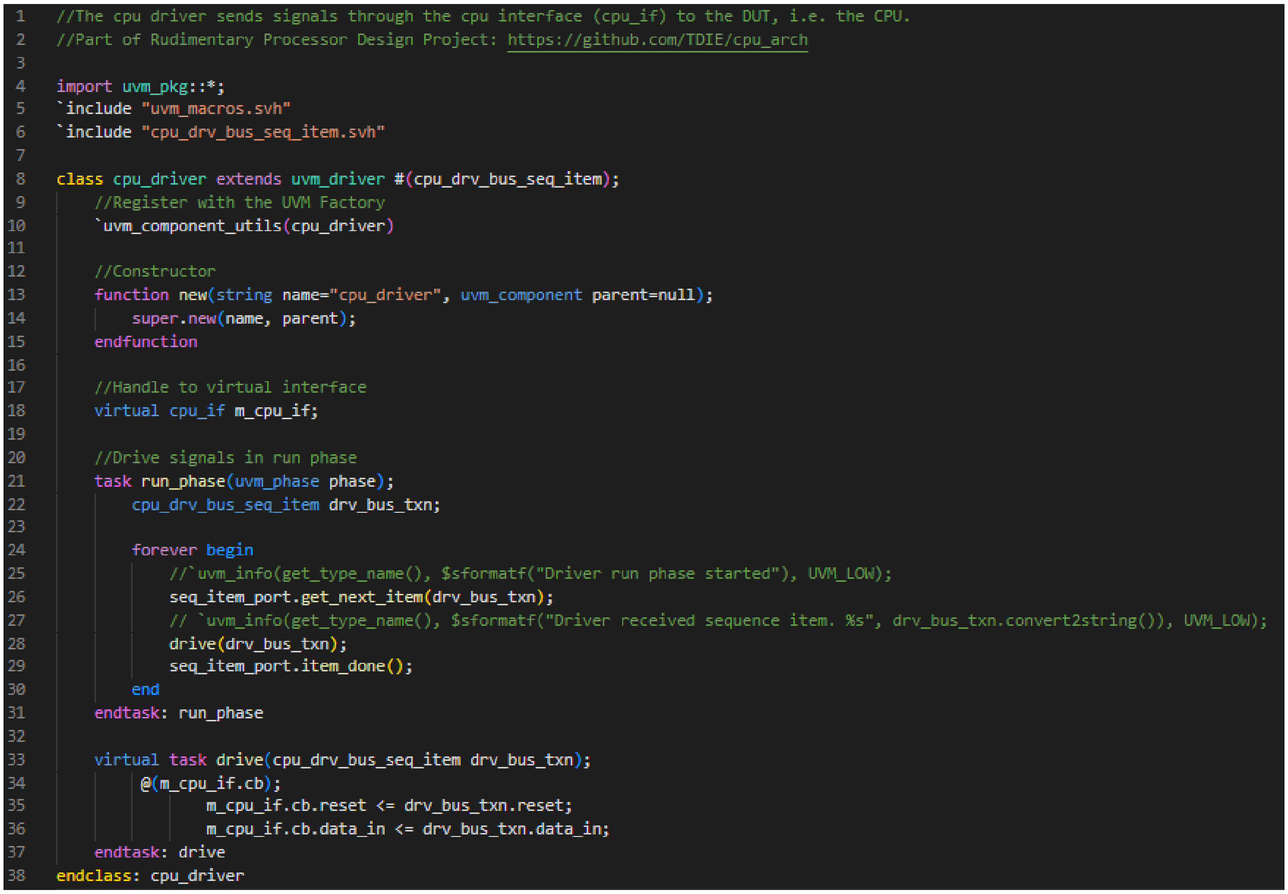

cpu_agent
[uvm_agent]
m_uvm_sequencer
[uvm_sequencer]
cpu_bus_seq
[uvm_sequence]
cpu_drv_bus_seq_item
[uvm_seq_item]
cpu_driver
cpu_if
[virt. Interface]
cpu_monitor
cpu_if
[virt. Interface]
ap
[uvm_analysis_port]
cpu_agent
+m_cpu_agent_config: cpu_agent_config
+m_cpu_driver: cpu_driver
+m_cpu_monitor: cpu_monitor
+ap #(cpu_mon_bus_seq_item): uvm_analysis_port
+m_sequencer #(cpu_drv_bus_seq_item): uvm_sequencer
+build_phase(uvm_phase phase): void
+connect_phase(uvm_phase phase): void
cpu_driver
+cpu_if: virtual cpu_if
+run_phase(uvm_phase phase): (task)
+drive(cpu_drv_bus_seq_item): (task)
cpu_monitor
+ap #(cpu_mon_bus_seq_item): uvm_analysis_port
+m_cpu_if: virtual cpu_if
+m_agent_config: cpu_agent_config
+cpu_bus_seq_item: mon_bus_txn
+build_phase(uvm_phase phase): void
+run_phase(uvm_phase phase): (task)
cpu_agent_config
+cpu_if: virtual cpu_if
+active: uvm_active_passive_enum

Figure 86 Overview and UML Diagram of the Agent.



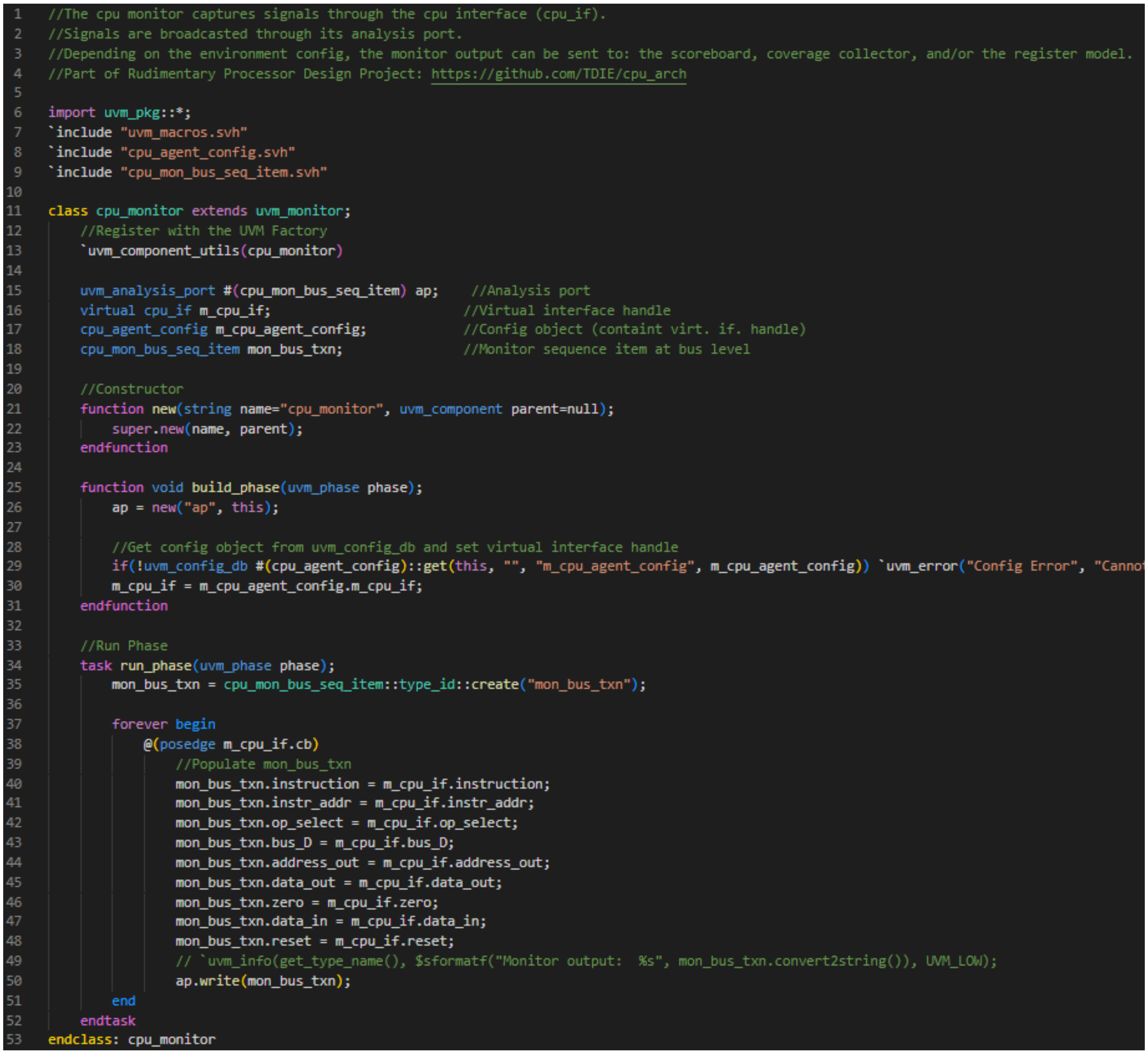
```
//The cpu agent contains the driver, monitor, and sequencer.
//Part of Rudimentary Processor Design Project: https://github.com/TDIE/cpu_arch

import uvm_pkg::*;
`include "uvm_macros.svh"
`include "cpu_driver.svh"
`include "cpu_monitor.svh"
// `include "cpu_bus_seq.svh"
// `include "cpu_drv_bus_seq_item.svh"
//`include "cpu_mon_bus_seq_item.svh"
//`include "cpu_agent_config.svh"

class cpu_agent extends uvm_agent;
    //Register with the UVM Factory
    `uvm_component_utils(cpu_agent)

    //Configuration Object
    cpu_agent_config m_cpu_agent_config;

    //Components
    cpu_driver m_cpu_driver;
    cpu_monitor m_cpu_monitor;
    uvm_analysis_port #(cpu_mon_bus_seq_item) ap;
    uvm_sequencer #(cpu_drv_bus_seq_item) m_sequencer;

    //Constructor
    function new(string name="cpu_agent", uvm_component parent=null);
        super.new(name, parent);
    endfunction

    //Build Phase: build the monitor (and sequencer+driver if config object is set accordingly)
    function void build_phase(uvm_phase phase);
        m_cpu_monitor = cpu_monitor::type_id::create("m_cpu_monitor", this);

        if(m_cpu_agent_config == null) begin
            if(!uvm_config_db #(cpu_agent_config)::get(this, "", "m_cpu_agent_config", m_cpu_agent_config)) `uvm_fatal(
        end
        if(m_cpu_agent_config.active == UVM_ACTIVE) begin
            m_cpu_driver = cpu_driver::type_id::create("m_cpu_driver", this);
            m_sequencer = uvm_sequencer #(cpu_drv_bus_seq_item)::type_id::create("m_sequencer", this);
        end
    endfunction: build_phase

    //Connect Phase: assign handle to monitor's analysis port and set virtual if handle
    function void connect_phase(uvm_phase phase);
        ap = m_cpu_monitor.ap;
        m_cpu_monitor.m_cpu_if = m_cpu_agent_config.m_cpu_if;

        if(m_cpu_agent_config.active == UVM_ACTIVE) begin
            m_cpu_driver.seq_item_port.connect(m_sequencer.seq_item_export);
            m_cpu_driver.m_cpu_if = m_cpu_agent_config.m_cpu_if;
        end
    endfunction
endclass: cpu_agent
```

Figure 87 Source code of the CPU agent (UVM).



```
//The cpu driver sends signals through the cpu interface (cpu_if) to the DUT, i.e. the CPU.
//Part of Rudimentary Processor Design Project: https://github.com/TDIE/cpu_arch

import uvm_pkg::*;
`include "uvm_macros.svh"
`include "cpu_drv_bus_seq_item.svh"

class cpu_driver extends uvm_driver #(cpu_drv_bus_seq_item);
    //Register with the UVM Factory
    `uvm_component_utils(cpu_driver)

    //Constructor
    function new(string name="cpu_driver", uvm_component parent=null);
        super.new(name, parent);
    endfunction

    //Handle to virtual interface
    virtual cpu_if m_cpu_if;

    //Drive signals in run phase
    task run_phase(uvm_phase phase);
        cpu_drv_bus_seq_item drv_bus_txn;

        forever begin
            //`uvm_info(get_type_name(), $sformatf("Driver run phase started"), UVM_LOW);
            seq_item_port.get_next_item(drv_bus_txn);
            // `uvm_info(get_type_name(), $sformatf("Driver received sequence item. %s", drv_bus_txn.convert2string()), UVM_LOW);
            drive(drv_bus_txn);
            seq_item_port.item_done();
        end
    endtask: run_phase

    virtual task drive(cpu_drv_bus_seq_item drv_bus_txn);
         @(m_cpu_if.cb);
                m_cpu_if.cb.reset <= drv_bus_txn.reset;
                m_cpu_if.cb.data_in <= drv_bus_txn.data_in;
    endtask: drive
endclass: cpu_driver
```

Figure 88 Source code of the CPU Driver (UVM).



```
//The cpu monitor captures signals through the cpu interface (cpu_if).
//Signals are broadcasted through its analysis port.
//Depending on the environment config, the monitor output can be sent to: the scoreboard, coverage collector, and/or the register model.
//Part of Rudimentary Processor Design Project: https://github.com/TDIE/cpu_arch

import uvm_pkg::*;
`include "uvm_macros.svh"
`include "cpu_agent_config.svh"
`include "cpu_mon_bus_seq_item.svh"

class cpu_monitor extends uvm_monitor;
    //Register with the UVM Factory
    `uvm_component_utils(cpu_monitor)

    uvm_analysis_port #(cpu_mon_bus_seq_item) ap;    //Analysis port
    virtual cpu_if m_cpu_if;                        //Virtual interface handle
    cpu_agent_config m_cpu_agent_config;            //Config object (containt virt. if. handle)
    cpu_mon_bus_seq_item mon_bus_txn;               //Monitor sequence item at bus level

    //Constructor
    function new(string name="cpu_monitor", uvm_component parent=null);
        super.new(name, parent);
    endfunction

    function void build_phase(uvm_phase phase);
        ap = new("ap", this);

        //Get config object from uvm_config_db and set virtual interface handle
        if(!uvm_config_db #(cpu_agent_config)::get(this, "", "m_cpu_agent_config", m_cpu_agent_config)) `uvm_error("Config Error", "Cannot
        m_cpu_if = m_cpu_agent_config.m_cpu_if;
    endfunction

    //Run Phase
    task run_phase(uvm_phase phase);
        mon_bus_txn = cpu_mon_bus_seq_item::type_id::create("mon_bus_txn");

        forever begin
            @(posedge m_cpu_if.cb)
                //Populate mon_bus_txn
                mon_bus_txn.instruction = m_cpu_if.instruction;
                mon_bus_txn.instr_addr = m_cpu_if.instr_addr;
                mon_bus_txn.op_select = m_cpu_if.op_select;
                mon_bus_txn.bus_D = m_cpu_if.bus_D;
                mon_bus_txn.address_out = m_cpu_if.address_out;
                mon_bus_txn.data_out = m_cpu_if.data_out;
                mon_bus_txn.zero = m_cpu_if.zero;
                mon_bus_txn.data_in = m_cpu_if.data_in;
                mon_bus_txn.reset = m_cpu_if.reset;
                // `uvm_info(get_type_name(), $sformatf("Monitor output:  %s", mon_bus_txn.convert2string()), UVM_LOW);
                ap.write(mon_bus_txn);
        end
    endtask
endclass: cpu_monitor
```

Figure 89 Source code of the CPU Monitor (UVM).

5.3.2.5 Sequence and Sequence Items

A single sequence will both initialize the processor and send input to it via driver sequence items. The monitor will capture the results from the CPU operations and send monitor sequence items to the scoreboard to verify correctness. UML diagrams for these sequence items as well as the sequence are shown below. The term “bus sequence” is chosen to differentiate it from a register sequence.

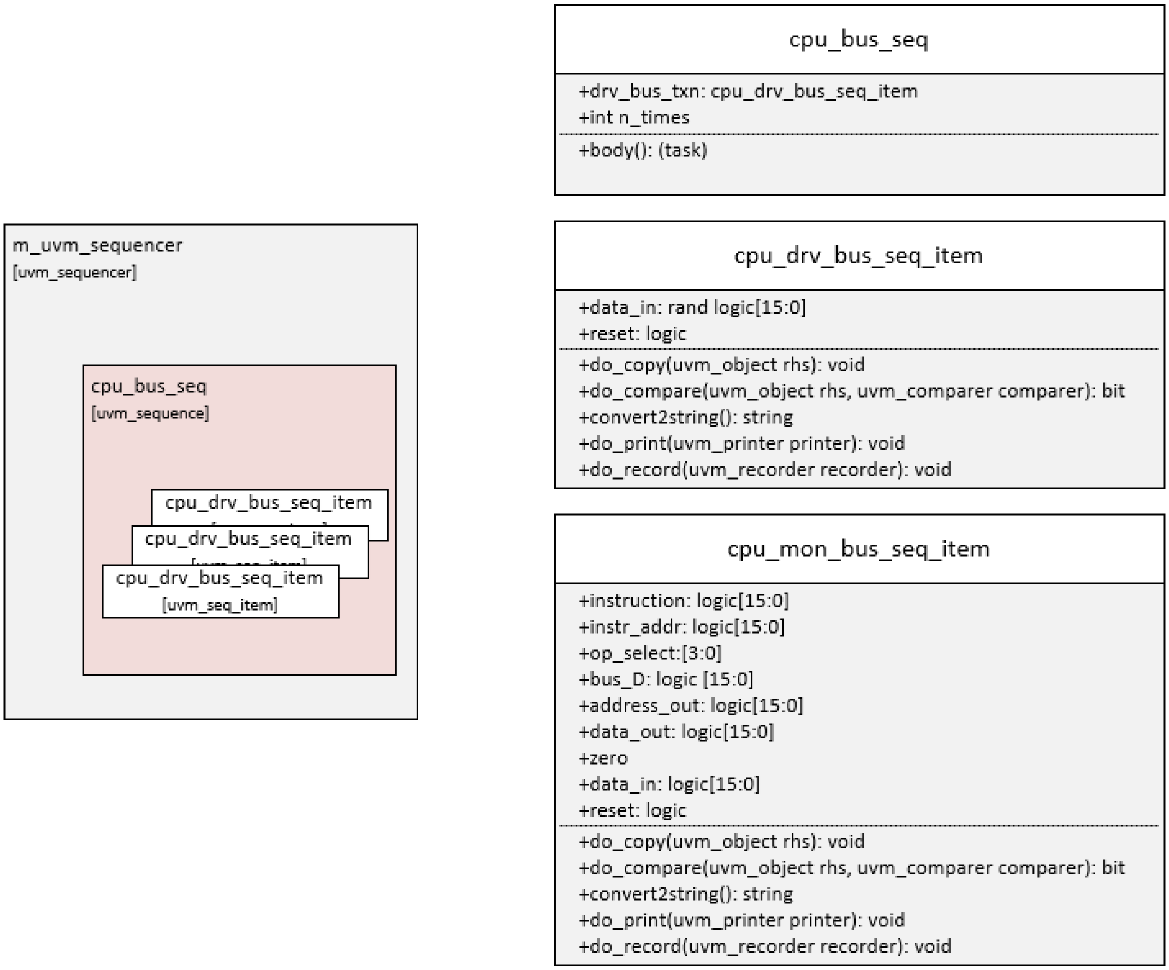

m_uvm_sequencer
[uvm_sequencer]
cpu_bus_seq
[uvm_sequence]
cpu_drv_bus_seq_item
cpu_drv_bus_seq_item
cpu_drv_bus_seq_item
[uvm_seq_item]
cpu_bus_seq
+drv_bus_txn: cpu_drv_bus_seq_item
+int n_times
+body(): (task)
cpu_drv_bus_seq_item
+data_in: rand logic[15:0]
+reset: logic
+do_copy(uvm_object rhs): void
+do_compare(uvm_object rhs, uvm_comparer comparer): bit
+convert2string(): string
+do_print(uvm_printer printer): void
+do_record(uvm_recorder recorder): void
cpu_mon_bus_seq_item
+instruction: logic[15:0]
+instr_addr: logic[15:0]
+op_select:[3:0]
+bus_D: logic [15:0]
+address_out: logic[15:0]
+data_out: logic[15:0]
+zero
+data_in: logic[15:0]
+reset: logic
+do_copy(uvm_object rhs): void
+do_compare(uvm_object rhs, uvm_comparer comparer): bit
+convert2string(): string
+do_print(uvm_printer printer): void
+do_record(uvm_recorder recorder): void

Figure 90 UML Diagrams of the sequence items.

5.3.2.6 Scoreboard

The scoreboard is connected to the monitor’s analysis port and implements its write() method. This is standard UVM practice (and an example of the observer OOP pattern). Upon receiving a sequence item, a copy is made first. The evaluate method then calls the predict method which calculates the expected outcome based on the copy of the monitor’s sequence item. The evaluator compares this predicted outcome to the sequence item copy. The scoreboard counts correct and incorrect results and prints out these numbers during the UVM’s report phase.

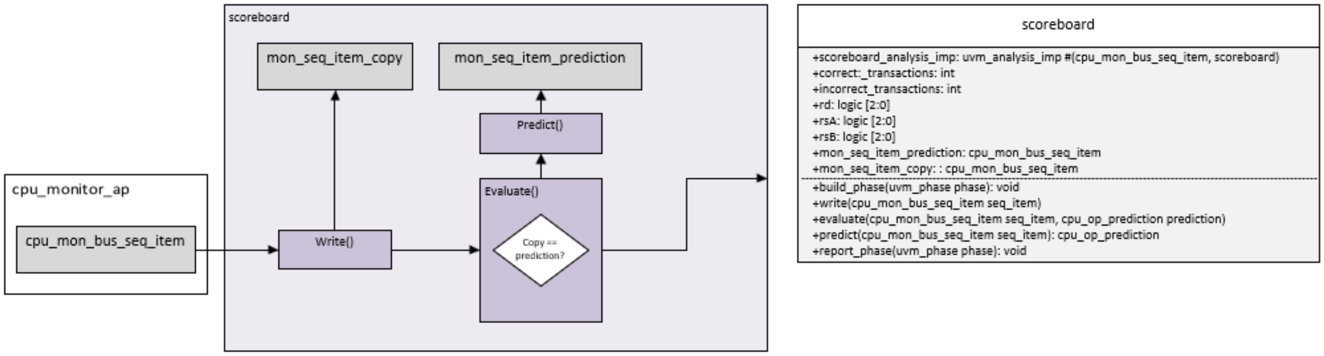
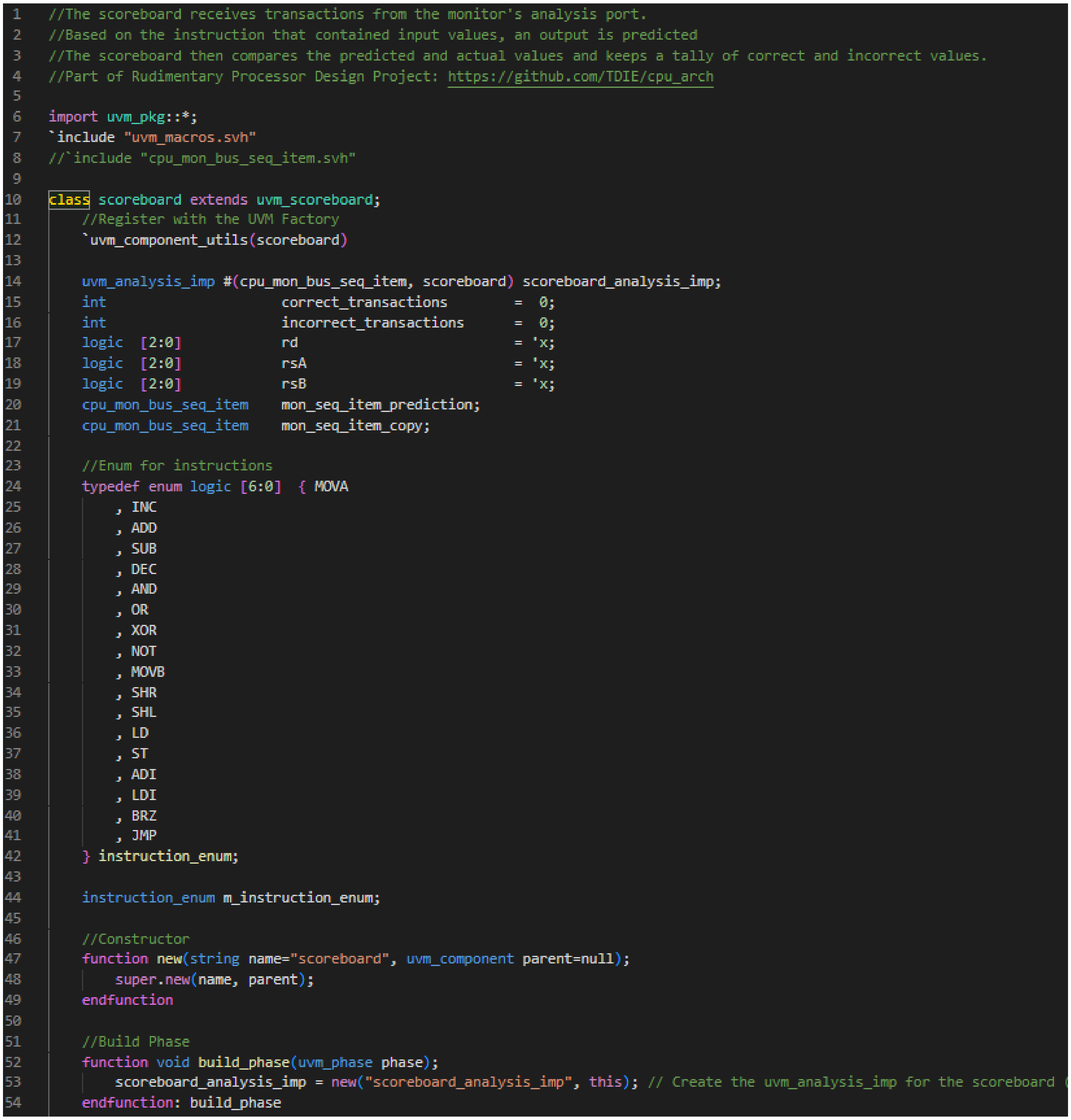
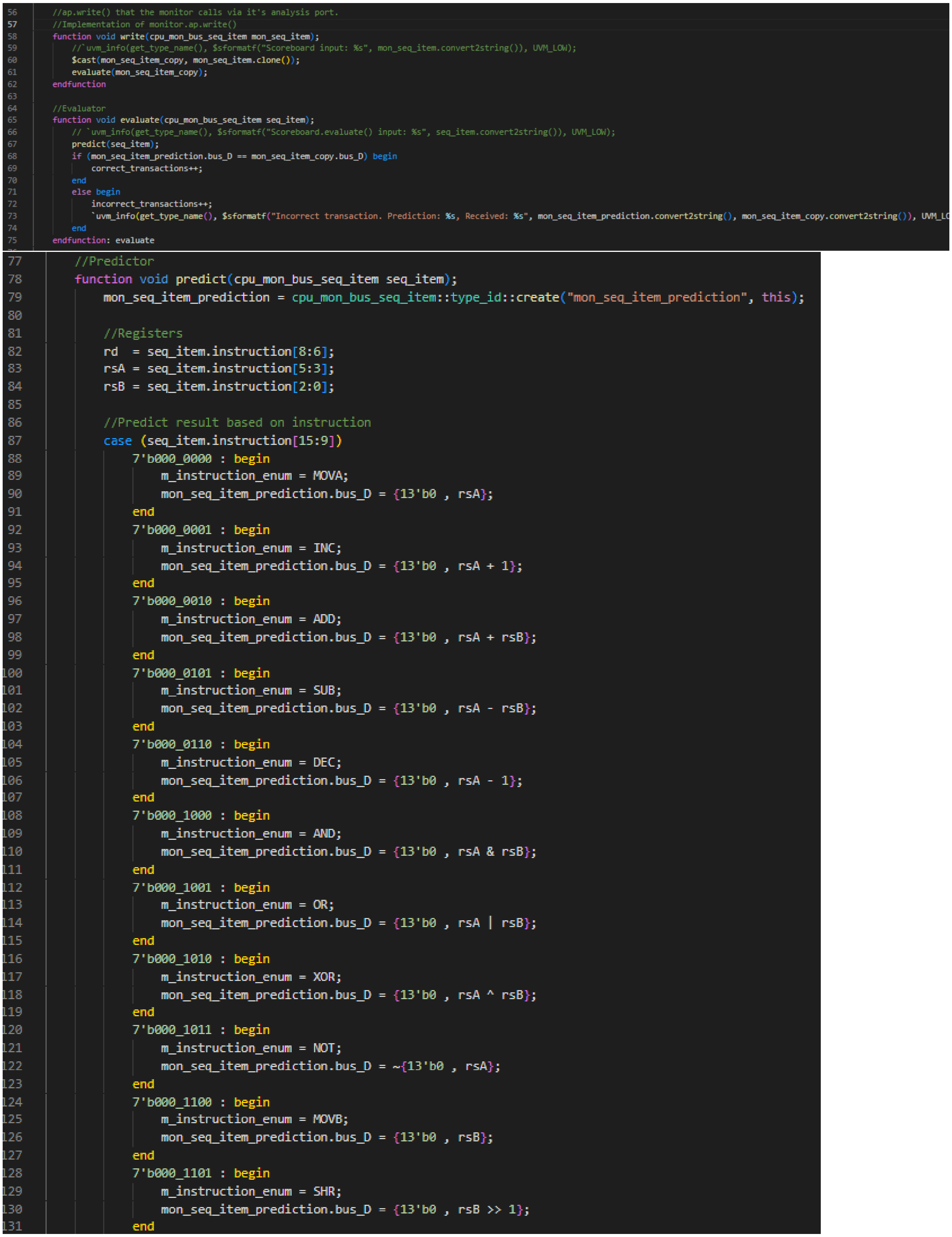

scoreboard
mon_seq_item_copy
mon_seq_item_prediction
Predict()
cpu_monitor_ap
cpu_mon_bus_seq_item
Write()
Evaluate()
Copy == prediction?
scoreboard
+scoreboard_analysis_imp: uvm_analysis_imp #(cpu_mon_bus_seq_item, scoreboard)
+correct:_transactions: int
+incorrect_transactions: int
+rd: logic [2:0]
+rsA: logic [2:0]
+rsB: logic [2:0]
+mon_seq_item_prediction: cpu_mon_bus_seq_item
+mon_seq_item_copy: : cpu_mon_bus_seq_item
+build_phase(uvm_phase phase): void
+write(cpu_mon_bus_seq_item seq_item)
+evaluate(cpu_mon_bus_seq_item seq_item, cpu_op_prediction prediction)
+predict(cpu_mon_bus_seq_item seq_item): cpu_op_prediction
+report_phase(uvm_phase phase): void

Figure 91Overview and UML Diagram of the Scoreboard.



```
//The scoreboard receives transactions from the monitor's analysis port.
//Based on the instruction that contained input values, an output is predicted
//The scoreboard then compares the predicted and actual values and keeps a tally of correct and incorrect values.
//Part of Rudimentary Processor Design Project: https://github.com/TDIE/cpu_arch

import uvm_pkg::*;
`include "uvm_macros.svh"
//`include "cpu_mon_bus_seq_item.svh"

class scoreboard extends uvm_scoreboard;
    //Register with the UVM Factory
    `uvm_component_utils(scoreboard)

    uvm_analysis_imp #(cpu_mon_bus_seq_item, scoreboard) scoreboard_analysis_imp;
    int                     correct_transactions        =  0;
    int                     incorrect_transactions      =  0;
    logic  [2:0]            rd                          = 'x;
    logic  [2:0]            rsA                         = 'x;
    logic  [2:0]            rsB                         = 'x;
    cpu_mon_bus_seq_item    mon_seq_item_prediction;
    cpu_mon_bus_seq_item    mon_seq_item_copy;

    //Enum for instructions
    typedef enum logic [6:0]  { MOVA
        , INC
        , ADD
        , SUB
        , DEC
        , AND
        , OR
        , XOR
        , NOT
        , MOVB
        , SHR
        , SHL
        , LD
        , ST
        , ADI
        , LDI
        , BRZ
        , JMP
    } instruction_enum;

    instruction_enum m_instruction_enum;

    //Constructor
    function new(string name="scoreboard", uvm_component parent=null);
        super.new(name, parent);
    endfunction

    //Build Phase
    function void build_phase(uvm_phase phase);
        scoreboard_analysis_imp = new("scoreboard_analysis_imp", this); // Create the uvm_analysis_imp for the scoreboard (
    endfunction: build_phase
```

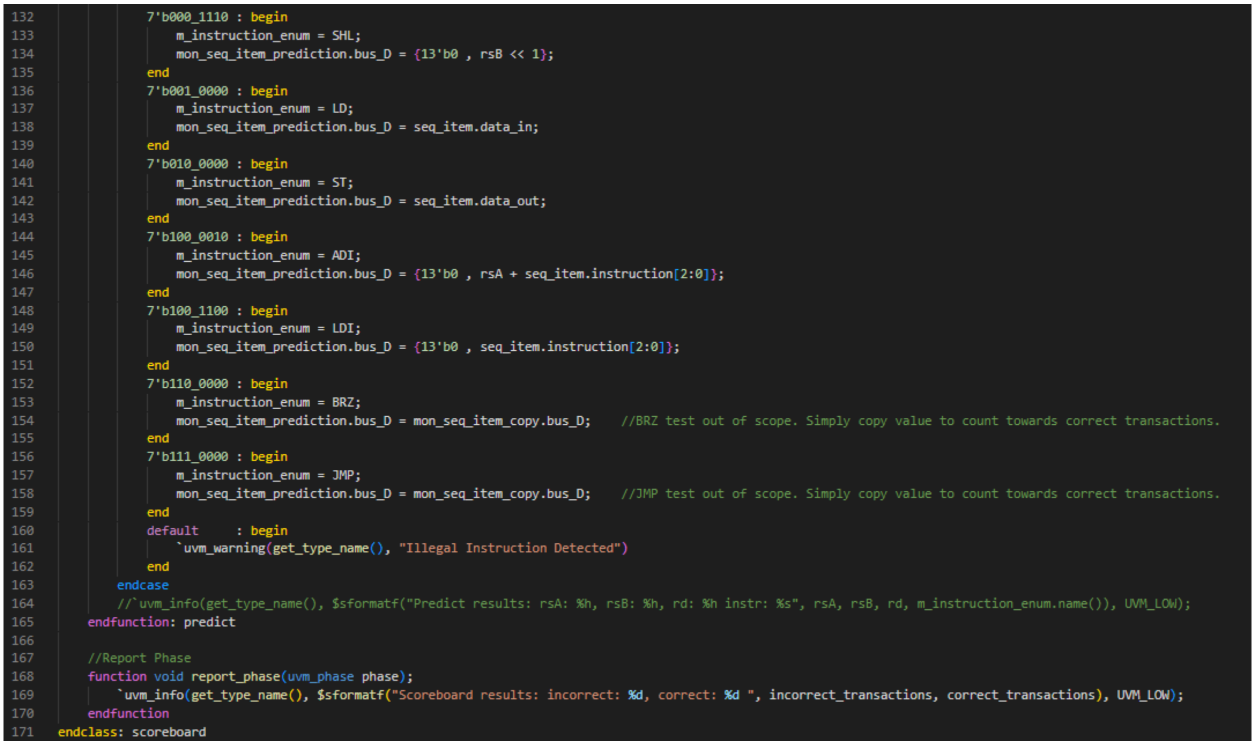
Figure 92 Source code for the scoreboard (1/3).



```
    //ap.write() that the monitor calls via it's analysis port.
    //Implementation of monitor.ap.write()
    function void write(cpu_mon_bus_seq_item mon_seq_item);
        //`uvm_info(get_type_name(), $sformatf("Scoreboard input: %s", mon_seq_item.convert2string()), UVM_LOW);
        $cast(mon_seq_item_copy, mon_seq_item.clone());
        evaluate(mon_seq_item_copy);
    endfunction

    //Evaluator
    function void evaluate(cpu_mon_bus_seq_item seq_item);
        // `uvm_info(get_type_name(), $sformatf("Scoreboard.evaluate() input: %s", seq_item.convert2string()), UVM_LOW);
        predict(seq_item);
        if (mon_seq_item_prediction.bus_D == mon_seq_item_copy.bus_D) begin
            correct_transactions++;
        end
        else begin
            incorrect_transactions++;
            `uvm_info(get_type_name(), $sformatf("Incorrect transaction. Prediction: %s, Received: %s", mon_seq_item_prediction.convert2string(), mon_seq_item_copy.convert2string()), UVM_L
        end
    endfunction: evaluate
```

```
    //Predictor
    function void predict(cpu_mon_bus_seq_item seq_item);
        mon_seq_item_prediction = cpu_mon_bus_seq_item::type_id::create("mon_seq_item_prediction", this);

        //Registers
        rd  = seq_item.instruction[8:6];
        rsA = seq_item.instruction[5:3];
        rsB = seq_item.instruction[2:0];

        //Predict result based on instruction
        case (seq_item.instruction[15:9])
            7'b000_0000 : begin
                m_instruction_enum = MOVA;
                mon_seq_item_prediction.bus_D = {13'b0 , rsA};
            end
            7'b000_0001 : begin
                m_instruction_enum = INC;
                mon_seq_item_prediction.bus_D = {13'b0 , rsA + 1};
            end
            7'b000_0010 : begin
                m_instruction_enum = ADD;
                mon_seq_item_prediction.bus_D = {13'b0 , rsA + rsB};
            end
            7'b000_0101 : begin
                m_instruction_enum = SUB;
                mon_seq_item_prediction.bus_D = {13'b0 , rsA - rsB};
            end
            7'b000_0110 : begin
                m_instruction_enum = DEC;
                mon_seq_item_prediction.bus_D = {13'b0 , rsA - 1};
            end
            7'b000_1000 : begin
                m_instruction_enum = AND;
                mon_seq_item_prediction.bus_D = {13'b0 , rsA & rsB};
            end
            7'b000_1001 : begin
                m_instruction_enum = OR;
                mon_seq_item_prediction.bus_D = {13'b0 , rsA | rsB};
            end
            7'b000_1010 : begin
                m_instruction_enum = XOR;
                mon_seq_item_prediction.bus_D = {13'b0 , rsA ^ rsB};
            end
            7'b000_1011 : begin
                m_instruction_enum = NOT;
                mon_seq_item_prediction.bus_D = ~{13'b0 , rsA};
            end
            7'b000_1100 : begin
                m_instruction_enum = MOVB;
                mon_seq_item_prediction.bus_D = {13'b0 , rsB};
            end
            7'b000_1101 : begin
                m_instruction_enum = SHR;
                mon_seq_item_prediction.bus_D = {13'b0 , rsB >> 1};
            end
```

Figure 93 Source code of the scoreboard (2/3)



```
            7'b000_1110 : begin
                m_instruction_enum = SHL;
                mon_seq_item_prediction.bus_D = {13'b0 , rsB << 1};
            end
            7'b001_0000 : begin
                m_instruction_enum = LD;
                mon_seq_item_prediction.bus_D = seq_item.data_in;
            end
            7'b010_0000 : begin
                m_instruction_enum = ST;
                mon_seq_item_prediction.bus_D = seq_item.data_out;
            end
            7'b100_0010 : begin
                m_instruction_enum = ADI;
                mon_seq_item_prediction.bus_D = {13'b0 , rsA + seq_item.instruction[2:0]};
            end
            7'b100_1100 : begin
                m_instruction_enum = LDI;
                mon_seq_item_prediction.bus_D = {13'b0 , seq_item.instruction[2:0]};
            end
            7'b110_0000 : begin
                m_instruction_enum = BRZ;
                mon_seq_item_prediction.bus_D = mon_seq_item_copy.bus_D;    //BRZ test out of scope. Simply copy value to count towards correct transactions.
            end
            7'b111_0000 : begin
                m_instruction_enum = JMP;
                mon_seq_item_prediction.bus_D = mon_seq_item_copy.bus_D;    //JMP test out of scope. Simply copy value to count towards correct transactions.
            end
            default     : begin
                `uvm_warning(get_type_name(), "Illegal Instruction Detected")
            end
        endcase
        //`uvm_info(get_type_name(), $sformatf("Predict results: rsA: %h, rsB: %h, rd: %h instr: %s", rsA, rsB, rd, m_instruction_enum.name()), UVM_LOW);
    endfunction: predict

    //Report Phase
    function void report_phase(uvm_phase phase);
        `uvm_info(get_type_name(), $sformatf("Scoreboard results: incorrect: %d, correct: %d ", incorrect_transactions, correct_transactions), UVM_LOW);
    endfunction
endclass: scoreboard
```

Figure 94 Source code of the scoreboard (3/3).

5.3.2.7 Coverage Collector

The coverage collector contains a cover group with several cover points. They track which instructions have been performed and which registers have been used. As stated earlier, the free EDA tool used for this project, Questa Sim, did not support cover groups, unfortunately, so this code was not tested and out of scope for this project.

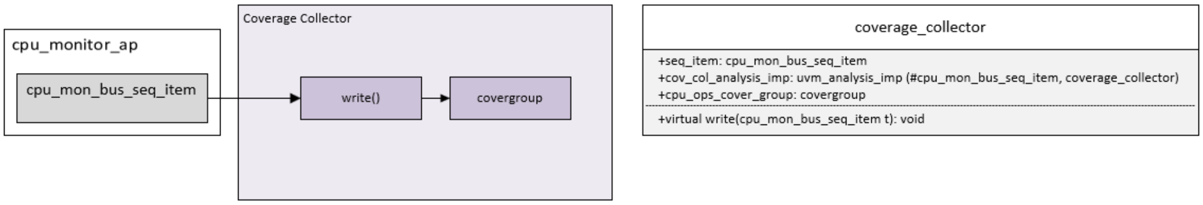
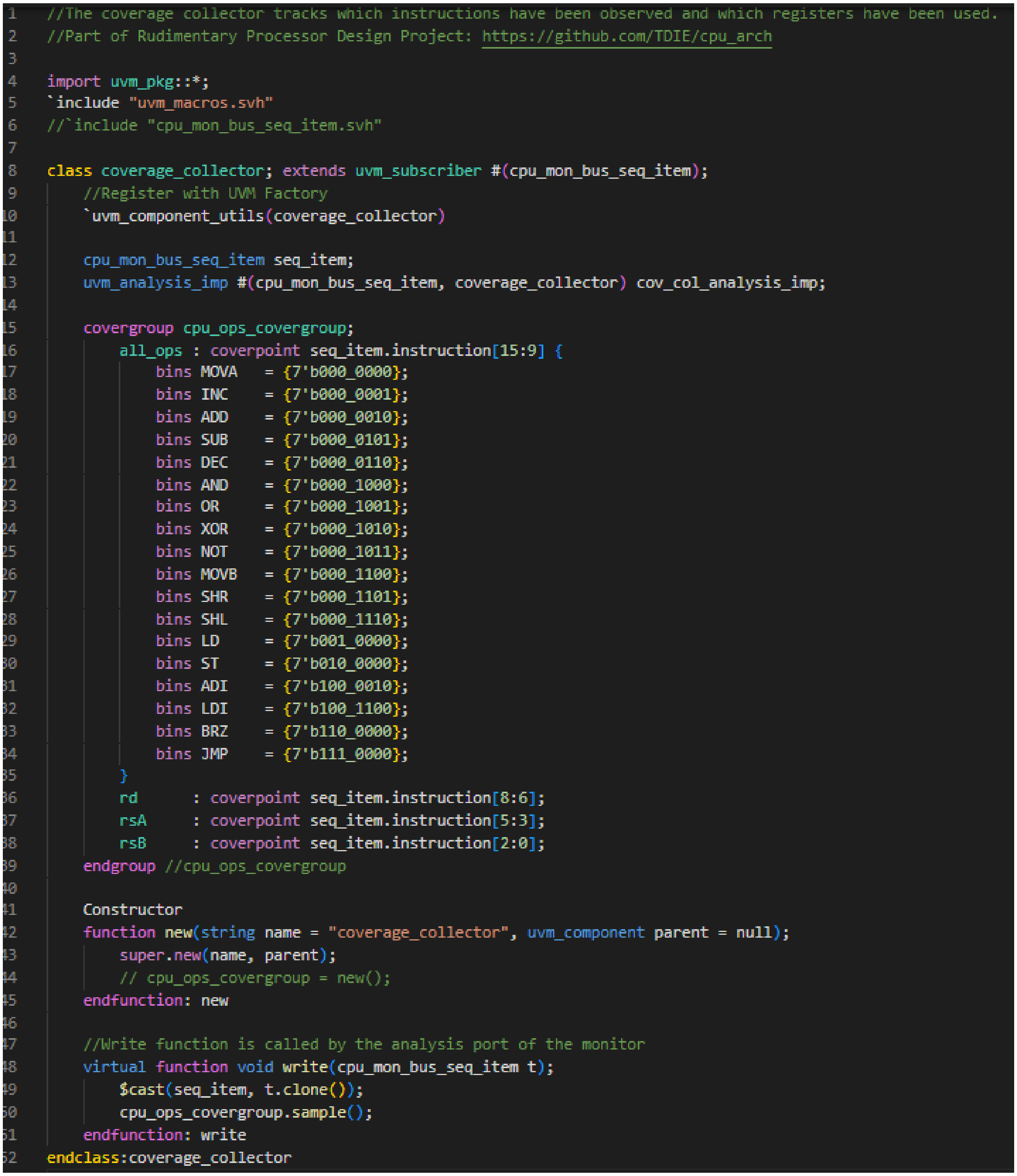

cpu_monitor_ap
cpu_mon_bus_seq_item
Coverage Collector
write()
covergroup
coverage_collector
+seq_item: cpu_mon_bus_seq_item
+cov_col_analysis_imp: uvm_analysis_imp (#cpu_mon_bus_seq_item, coverage_collector)
+cpu_ops_cover_group: covergroup
+virtual write(cpu_mon_bus_seq_item t): void

Figure 95 Overview and UML Diagram of the Coverage Collector.



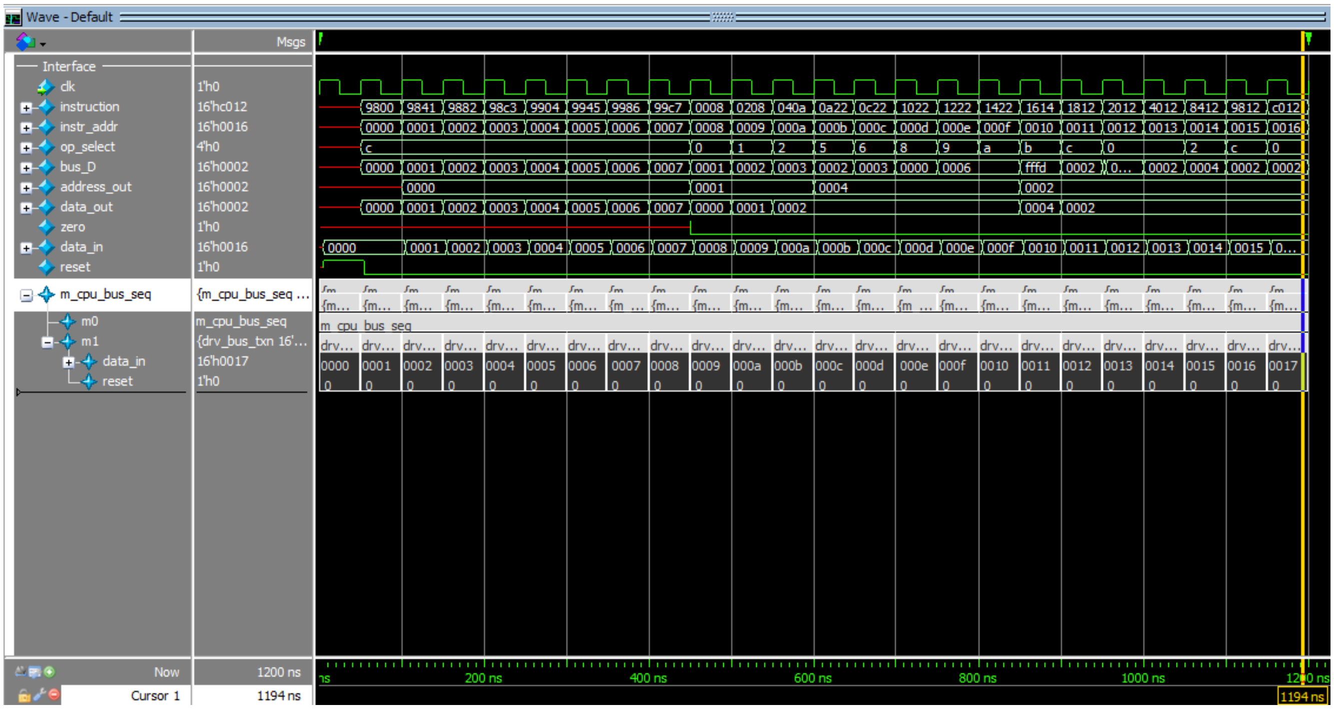
```
//The coverage collector tracks which instructions have been observed and which registers have been used.
//Part of Rudimentary Processor Design Project: https://github.com/TDIE/cpu_arch

import uvm_pkg::*;
`include "uvm_macros.svh"
//`include "cpu_mon_bus_seq_item.svh"

class coverage_collector; extends uvm_subscriber #(cpu_mon_bus_seq_item);
    //Register with UVM Factory
    `uvm_component_utils(coverage_collector)

    cpu_mon_bus_seq_item seq_item;
    uvm_analysis_imp #(cpu_mon_bus_seq_item, coverage_collector) cov_col_analysis_imp;

    covergroup cpu_ops_covergroup;
        all_ops : coverpoint seq_item.instruction[15:9] {
            bins MOVA   = {7'b000_0000};
            bins INC    = {7'b000_0001};
            bins ADD    = {7'b000_0010};
            bins SUB    = {7'b000_0101};
            bins DEC    = {7'b000_0110};
            bins AND    = {7'b000_1000};
            bins OR     = {7'b000_1001};
            bins XOR    = {7'b000_1010};
            bins NOT    = {7'b000_1011};
            bins MOVB   = {7'b000_1100};
            bins SHR    = {7'b000_1101};
            bins SHL    = {7'b000_1110};
            bins LD     = {7'b001_0000};
            bins ST     = {7'b010_0000};
            bins ADI    = {7'b100_0010};
            bins LDI    = {7'b100_1100};
            bins BRZ    = {7'b110_0000};
            bins JMP    = {7'b111_0000};
        }
        rd      : coverpoint seq_item.instruction[8:6];
        rsA     : coverpoint seq_item.instruction[5:3];
        rsB     : coverpoint seq_item.instruction[2:0];
    endgroup //cpu_ops_covergroup

    Constructor
    function new(string name = "coverage_collector", uvm_component parent = null);
        super.new(name, parent);
        // cpu_ops_covergroup = new();
    endfunction: new

    //Write function is called by the analysis port of the monitor
    virtual function void write(cpu_mon_bus_seq_item t);
        $cast(seq_item, t.clone());
        cpu_ops_covergroup.sample();
    endfunction: write
endclass:coverage_collector
```

Figure 96 Source code for the coverage collector.

5.3.3 Test Results

Test results of the UVM base test are shown below in Figure 98. A waveform plot of the values on the interface suggests correct operation. Correctness is indeed confirmed by the output of the scoreboard: all 24 instructions produced the correct result! As stated earlier: jump and branch were out of scope for the UVM testbench and were already verified elsewhere so they got an automatic pass on this test.

The one incorrect transaction listed is the one on the very first rising edge of the clock. Values are unknown at that point because the reset signal is set to propagate 5ns after the rising edge of the clock which is dictated in the interface's clocking block. This one "incorrect" transaction is therefore to be expected and does not indicated incorrect operation.

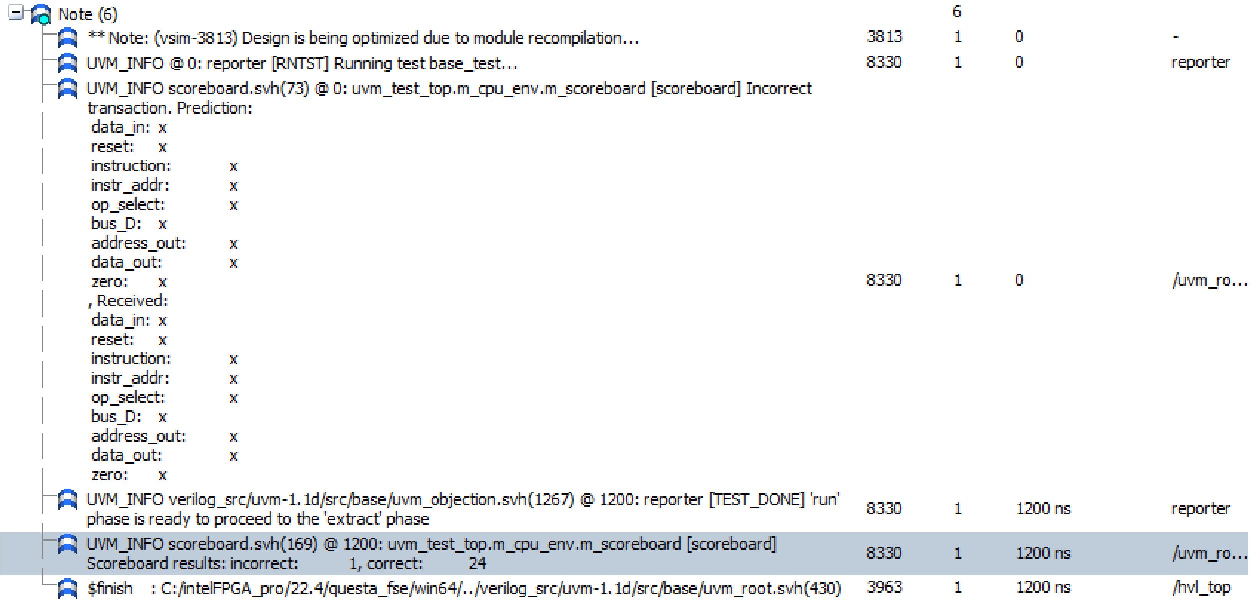


Wave - Default

```
                 Msgs
Interface
  clk            1'h0
  instruction    16'hc012
  instr_addr     16'h0016
  op_select      4'h0
  bus_D          16'h0002
  address_out    16'h0002
  data_out       16'h0002
  zero           1'h0
  data_in        16'h0016
  reset          1'h0
m_cpu_bus_seq    {m_cpu_bus_seq ...
  m0             m_cpu_bus_seq
  m1             {drv_bus_txn 16'...
    data_in      16'h0017
    reset        1'h0

Now       1200 ns
Cursor 1  1194 ns
```

9800 9841 9882 98c3 9904 9945 9986 99c7 0008 0208 040a 0a22 0c22 1022 1222 1422 1614 1812 2012 4012 8412 9812 c012
0000 0001 0002 0003 0004 0005 0006 0007 0008 0009 000a 000b 000c 000d 000e 000f 0010 0011 0012 0013 0014 0015 0016
c 0 1 2 5 6 8 9 a b c 0 2 c 0
0000 0001 0002 0003 0004 0005 0006 0007 0001 0002 0003 0002 0003 0000 0006 fffd 0002 0... 0002 0004 0002 0002
0000 0001 0004 0002
0000 0001 0002 0003 0004 0005 0006 0007 0000 0001 0002 0004 0002
0000 0001 0002 0003 0004 0005 0006 0007 0008 0009 000a 000b 000c 000d 000e 000f 0010 0011 0012 0013 0014 0015 0...
{m...
m_cpu_bus_seq
drv...
0000 0001 0002 0003 0004 0005 0006 0007 0008 0009 000a 000b 000c 000d 000e 000f 0010 0011 0012 0013 0014 0015 0016 0017
0
ns
200 ns
400 ns
600 ns
800 ns
1000 ns
1200 ns
1194 ns

Figure 97 Data on the cpu_if during sample program execution.



```
Note (6)                                                                                          6
  ** Note: (vsim-3813) Design is being optimized due to module recompilation...          3813    1     0           -
  UVM_INFO @ 0: reporter [RNTST] Running test base_test...                               8330    1     0           reporter
  UVM_INFO scoreboard.svh(73) @ 0: uvm_test_top.m_cpu_env.m_scoreboard [scoreboard] Incorrect
  transaction. Prediction:
   data_in: x
   reset:    x
   instruction:        x
   instr_addr:         x
   op_select:          x
   bus_D:    x
   address_out:        x
   data_out:           x
   zero:     x                                                                           8330    1     0           /uvm_ro...
  , Received:
   data_in: x
   reset:    x
   instruction:        x
   instr_addr:         x
   op_select:          x
   bus_D:    x
   address_out:        x
   data_out:           x
   zero:     x
  UVM_INFO verilog_src/uvm-1.1d/src/base/uvm_objection.svh(1267) @ 1200: reporter [TEST_DONE] 'run'
  phase is ready to proceed to the 'extract' phase                                       8330    1     1200 ns     reporter
  UVM_INFO scoreboard.svh(169) @ 1200: uvm_test_top.m_cpu_env.m_scoreboard [scoreboard]
  Scoreboard results: incorrect:          1, correct:        24                          8330    1     1200 ns     /uvm_ro...
  $finish    : C:/intelFPGA_pro/22.4/questa_fse/win64/../verilog_src/uvm-1.1d/src/base/uvm_root.svh(430)   3963    1     1200 ns     /hvl_top
```

Figure 98 Test Results of the UVM base_test.

6 FUTURE IMPROVEMENTS AND LESSONS LEARNED

This project was purely educational. The ISA and micro-architecture were kept simple to make the scope manageable and compress the schedule of the project to one to two quarters of evening and weekend work. This section describes several enhancements that would improve the processor's performance as well as lessons learned during this project.

6.1 POTENTIAL DESIGN ENHANCEMENTS

The architecture of this processor favored simplicity over performance. Its performance is too limited for practical applications. Low-cost microcontrollers are available that would beat its performance and functionality handsomely. Several improvements are listed below.

6.1.1 HW Improvements

- Write functionality for the instruction memory so different programs can be loaded.
- Status and Configuration Registers
- Watchdog Timer
- ALU:
  - overflow handling
  - signed arithmetic support
- Include (multi-level) Cache memory
- Branch prediction
- Support more instructions (e.g., a Floating Point Unit, FPU)
- Power management (sleep/awake/power levels)
- Support for Interrupts
- Include a Stack Pointer
- Pipelined architecture
- Super scalar architecture
- Broader functionality: ADC/DAC, UART, I2C, SPI, CAN bus, APB, etc.
- Dedicated AI support(e.g., neural network accelerator, Multiply and Accumulators)

6.1.2 Software/Firmware improvements:

- ISA Improvements:
  - Support an actual ISA: x86, RISC-V, ARMv8, etc.
  - Assembler or compiler to create SW for the processor (links back to standard ISA support).

The inclusion, or absence, of these features would depend on the targeted product, embedded vs automotive as well as market, consumer vs defense.

Furthermore, it would be beneficial to implement the design on an FPGA and provide the opportunity to step through each clock cycle with register values linked to LEDs or 7 segment displays. Again, this was excluded due to time constraints.

Nevertheless, the project's goals were fully achieved:

- Specify the micro-architecture for the processor.

- Design the RTL implementation of this micro-architecture (including implementation in Verilog).
- Design the system-level testbench architecture (using the UVM/SystemVerilog).
- Verify functional behavior pre-Si (simulation).

6.2 LESSONS LEARNED

This project allowed me to work cross functional. There were deliverables related to:

- Micro-architecture
- RTL Design
- Verification Architecture
- Verification Engineering
- Project Management

For future projects I will use the following lessons learned:

- Look into an affordable, more capable EDA tool that supports:
    - Constraint random input (.randomize()).
    - Cover groups
- Make testbench fully emulation compatible.
- Consider RTL Design or Functional Verification only to limit project timeline.
- Make even more use of git and GitHub’s features.

REFERENCES

[1] M. M. Mano and C. R. Kimo, Logic and Computer Design Fundamentals, Pearson Education, 2008.

[2] T. Diederen, "Github," June 2023. [Online]. Available: https://github.com/TDIE/BCD_UVM_Testbench.git.

[3] Mentor Graphics (Siemens), UVM Cookbook, verificationacadamy.com.