Rudimentary Processor Architecture

Design and pre-Silicon verification of a simple processor

Tom DIEDEREN

2023

Revision History

|  |  |  |
| --- | --- | --- |
| **Date** | **Version** | **Comments** |
| Aug 2023 | 0.1 | Added document layout and high-level top layer HW and testbench architecture |
| Sept 2023 | 0.3 | Initial HW and unit testbench architecture definition completed |
| Oct 2023 | 0.5 | HW RTL written, unit tests completed |
|  | 0.7 | UVM System testbench architecture completed |
|  | 0.8 | UVM tests completed, full functional coverage |
|  | 1.0 | Included lessons learned and future improvements |
|  |  |  |

Table of Contents

[List of Figures 4](#_Toc145593696)

[List of Tables 5](#_Toc145593697)

[Acronyms 6](#_Toc145593698)

[Summary 7](#_Toc145593699)

[1 Requirements & Specifications 7](#_Toc145593700)

[1.1 Datapath 7](#_Toc145593701)

[1.1.1 Arithmetic Logic Unit and Shifter 7](#_Toc145593702)

[1.1.2 Busses 7](#_Toc145593703)

[1.2 Control 8](#_Toc145593704)

[1.3 Instruction Set Architecture 8](#_Toc145593705)

[1.3.1 Registers 8](#_Toc145593706)

[1.3.2 Instruction Formats 8](#_Toc145593707)

[1.3.3 Instructions 9](#_Toc145593708)

[1.4 Power, Performance, and Area 10](#_Toc145593709)

[1.4.1 Power 10](#_Toc145593710)

[1.4.2 Performance and Area 10](#_Toc145593711)

[1.5 Pre-Si Verification 10](#_Toc145593712)

[2 Micro Architecture 12](#_Toc145593713)

[2.1 Top level 12](#_Toc145593714)

[2.1.1 Data Path 12](#_Toc145593715)

[2.1.2 Control Unit 16](#_Toc145593716)

[3 RTL Design 19](#_Toc145593717)

[3.1 Open issues (delete when done) 19](#_Toc145593718)

[3.2 Top Level 20](#_Toc145593719)

[3.2.1 Datapath 20](#_Toc145593720)

[3.2.2 Control Unit 27](#_Toc145593721)

[4 Verification 28](#_Toc145593722)

[4.1 Unit tests (Verilog testbenches) 28](#_Toc145593723)

[4.1.1 Datapath 28](#_Toc145593724)

[4.1.2 Control Unit 42](#_Toc145593725)

[4.2 Systel Level Test (UVM) 42](#_Toc145593726)

[4.2.1 Verification Strategy 42](#_Toc145593727)

[4.2.2 Testbench Architecture Overview 42](#_Toc145593728)

[4.2.3 Interface(s) 42](#_Toc145593729)

[4.2.4 Test(s) 42](#_Toc145593730)

[4.2.5 Use Cases & Transactions 42](#_Toc145593731)

[4.2.6 Scoreboard 42](#_Toc145593732)

[4.2.7 Functional Coverage 42](#_Toc145593733)

[4.2.8 Transactions 42](#_Toc145593734)

[4.2.9 Score (scoreboard) (combine with functional coverage chapter?) 42](#_Toc145593735)

[5 Future improvements and lessons learned 43](#_Toc145593736)

[5.1 Potential improvements (outside current scope) 43](#_Toc145593737)

[5.2 Lessons Learned 43](#_Toc145593738)

[6 References 46](#_Toc145593739)

[Appendix I Control Unit – Detailed View 47](#_Toc145593740)

[Appendix II 49](#_Toc145593741)

# List of Figures

[Figure 1 The design contains 8, 16-bit general purpose registers. 9](#_Toc145593742)

[Figure 2 16-bit program counter 9](#_Toc145593743)

[Figure 3 The data RAM has a 16-bit word size. 9](#_Toc145593744)

[Figure 4 Instructions are fetched from ROM. 9](#_Toc145593745)

[Figure 5 Register Instruction Format (R-type) 9](#_Toc145593746)

[Figure 6 Immediate Instruction Format (I-type) 10](#_Toc145593747)

[Figure 7 Jump / Branch Instruction Format 10](#_Toc145593748)

[Figure 8 Questions to answer in this section (verificationacademy.com). Delete when section is finished. 12](#_Toc145593749)

[Figure 9 Top Level Overview of the Datapath 14](#_Toc145593750)

[Figure 10 Detailed Overview of the Datapath 15](#_Toc145593751)

[Figure 11 EU Architecture 16](#_Toc145593752)

[Figure 12 Architecture of the register file 17](#_Toc145593753)

[Figure 13 Micro architecture of the Control Unit, right hand side connects to the data path shown in Figure 10 17](#_Toc145593754)

[Figure 14 State Diagram of the Program Counter 18](#_Toc145593755)

[Figure 15 Instruction Decoder Logic 20](#_Toc145593756)

[Figure 16 Overview of the architecture of the Execution Unit as shown in2.2.1 on page 13 21](#_Toc145593757)

[Figure 17 Top level RTL of the Execution Unit 22](#_Toc145593758)

[Figure 18 RTL design of the arithmetic part of the execution unit. 23](#_Toc145593759)

[Figure 19 RTL design of the logic part of the Execution Unit 23](#_Toc145593760)

[Figure 20 RTL design of the shifter part of the Execution Unit 24](#_Toc145593761)

[Figure 21 High level overview of the register file (as shown in section 2.2.2) 25](#_Toc145593762)

[Figure 22 Verilog description of the register file: ports, internal signals, and generate block for the registers. 26](#_Toc145593763)

[Figure 23 Verilog description of the register file: output multiplexers. 27](#_Toc145593764)

[Figure 24 RTL for a single register. 28](#_Toc145593765)

[Figure 25 Top level test bench for the Execution Unit 30](#_Toc145593766)

[Figure 26 Top level test bench for the Execution Unit (continued) 31](#_Toc145593767)

[Figure 27 Simulation results for the Execution Unit (waveform) 32](#_Toc145593768)

[Figure 28 Simulation results for the Execution Unit (text) 32](#_Toc145593769)

[Figure 29 RTL of the unit test bench for the Arithmetic Unit. 33](#_Toc145593770)

[Figure 30 RTL of the unit test bench for the Arithmetic Unit (continued). 34](#_Toc145593771)

[Figure 31 Simulation results for the Arithmetic Unit (waveform) 35](#_Toc145593772)

[Figure 32 Simulation results for the Arithmetic Unit (text) 35](#_Toc145593773)

[Figure 33 Simulation results for the Logic Unit (waveform) 36](#_Toc145593774)

[Figure 34 Simulation results for the Logic Unit (text) 36](#_Toc145593775)

[Figure 35 Test cases for the logic testbench (Verilog) 37](#_Toc145593776)

[Figure 36 Simulation results for the shifter (waveform) 38](#_Toc145593777)

[Figure 37 Simulation results for the shifter (text) 38](#_Toc145593778)

[Figure 38 Shifter testbench (Verilog) 39](#_Toc145593779)

[Figure 39 Verilog description of the testbench for the register file: IO, module instantiation, and monitor. 40](#_Toc145593780)

[Figure 40 Verilog description of the testbench for the register file: test stimuli. 41](#_Toc145593781)

[Figure 41 Test results for the top layer of the Register File (wave). 42](#_Toc145593782)

[Figure 42 Test results for the top layer of the Register File (text). 42](#_Toc145593783)

[Figure 43 Testbench results for an individual register (wave) 42](#_Toc145593784)

[Figure 44 Testbench results for an individual register (text). 43](#_Toc145593785)

# List of Tables

[Table 1 Acronyms used in this document 7](#_Toc145593786)

[Table 2 Instruction Specification of the Design 10](#_Toc145593787)

[Table 3 control bits for each instruction category 19](#_Toc145593788)

[Table 4 Logic Formula's for Datapath Control Bits 19](#_Toc145593789)

# Acronyms

Table 1 Acronyms used in this document

|  |  |  |
| --- | --- | --- |
| Acronym | Unabbreviated | Description |
| 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) |
| 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. |
| 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. |
| ALU | Arithmetic Logic Unit | Digital circuit that can perform arithmetic and logic operations on binary integers. |
| TDP |  |  |
| IPC | Instructions per Cycle |  |
| FPU | Floating Point Unit |  |
|  |  |  |
|  |  |  |
|  |  |  |

# Summary

The goals of this project were:

* implement a simple processor architecture in Verilog.
* design a UVM testbench for it.
* verify its behavior (pre-Si / in simulation)

The project goals were met and resulted in increased Verilog proficiency, greater familiarity with the UVM, as well as enhanced knowledge of computer architecture for the author. Although the high-level architecture is based on chapter 9 of [1], the microarchitecture design, Verilog code, verification strategy, UVM testbench design and code, as well as pre-Si verification results, are all newly created by the author and not part of [1] or any other source.

Lastly, this project was done after work hours so to make completion in a few months feasible, the scope had to be limited. This means that the goal of completion on time was sometimes chosen over design option that would increase performance or efficiency which obviously wouldn’t be the case had this been a commercial project.

**Keywords/Skills:**

Verilog, System Verilog, UVM, Micro-Architecture, Hardware Design, Logic Design, Verification, Validation, Python, TCL, Simulation, Questa Sim, FPGA, Instruction Set Architecture.

# Requirements & Specifications

## Datapath

The datapath contains an ALU, shifter, register file and three busses which are described in 1.1.2.

### Arithmetic Logic Unit and Shifter

* Support the following micro-operations:
  + Add the content of two registers.
  + Subtract the content of two registers.
  + Increment the content of a register.
  + Decrement the content of a register.
  + Shift the content of a register right.
  + Shift the content of a register left.
  + Bitwise AND
  + Bitwise OR
  + Bitwise XOR
  + Invert (not / 1’s complement)
* HW described in Verilog
* 16-bit bus width in Verilog (parameterized for re-use)

### Busses

The design contains 3, 16-bit, busses: bus A and bus B. 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.

## Control

* Non-pipelined
* Scalar design
* No interrupts
* No power management (on/off only)

### Instruction memory

The Instruction Memory will be combination to facilitate single cycle execution (fetch, decode, execute). If the scope of this project allowed for back end implementation, this memory would be some kind of reprogrammable ROM. To keep the scope of this project manageable, the instruction memory will hold a single sample program.

## Instruction Set Architecture

### Registers

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

* 8 general purpose registers (0-7 within a register file)
* program counter
* Data memory, RAM,
* 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 8, 16-bit general purpose 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 has a 16-bit word size.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 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. Size for verification? 1MB for validation for example? Add memory map showing physical addresses in hex format?

### 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, only 8 registers can be addressed in the register file. This is sufficient to demonstrate functionality but most likely inadequate for any practical applications. A potential future upgrade, as mentioned in section 5.1, would be to design for a more realistic instruction set, e.g. RISC-V. To keep the scope, and schedule, of this project manageable, the simple ISA was chosen.

### Instructions

The architecture supports 19 basic instructions. The 3 MSBs of the op code define the type of instruction. The 4 LSBs further specify the exact instruction of a certain type, e.g. add vs subtract. The instruction formats and registers mentioned in Table 2 were described in section 1.3.2.

//Remove branch on negative (unsigned arithmetic only to limit scope and make project feasible in a few months of after-hours work)

Table 2 Instruction Specification of the Design //Update imm to specific register, remove sign extension, rename AD, LDI typo



## Power, Performance, and Area

Since this project was done after hours and by one-person the scope had to be limited. The decision was made to only focus on front-end design, i.e. the micro-architecture, RTL design, testbench architecture, and test implementation (pre-Si). Physical design and backend implementation considerations such as: FGPA vs ASIC, floorplan, die size, etc., are out of scope.

### Power

To keep the scope manageable, no power requirements are put on the design, e.g. through UPF. The HW only has two power states: on and off. No sleep or low power states were implemented and there were no constraints with regards to projected power, TDP, etc.

### Performance and Area

Due to its educational nature, no specific performance numbers, e.g. clock speed or IPC were specified. There are also no limits on area aspects such as number of gates used.

## Pre-Si Verification

Verification will be done in two stages:

1. using basic Verilog testbenches per functional block.
2. UVM based testbench for full design.

Formal verification is out of scope for this project. Furthermore, the verification architecture should:

* Use and enhance UVM test bench I designed and built earlier [2]
* Use the following EDA SW: Questa Sim (free Intel FPGA version) or EDA Playground? Check limitations of each.
  + Free simulator that can handle:
    - Constrained randomized inputs? (Not part of free Questa)
    - Unlimited sequence items? (36 items seemed the limit in free Questa during [2])

Create Verification Plan

* (Verification Architecture Doc.  
   -> section 4.2.
* Strategy: one environment, one agent, one interface, etc.
* Build order: phase 1, phase 2. Etc.
* UVM version.
* What will be applied to the DUT
* What will the scoreboard check
* What will be checked with assertions
* Functional Coverage
* Version Control (Git)
* Package with separate .svh files for test, environment, etc.

The UVM testbench (System Verilog) must include or make use of:

* + The UVM Register Abstraction Layer
  + Layered / virtual sequence.
  + Constrained random stimulus generation.
  + Functional coverage checks
    - Cover groups
    - Cover points
    - Bins, cross. Binsof, ignore\_bins, intersect
  + Test Cases:
    - Execute all operations after reset
    - All 0’s on one EU input
    - All 1’s on one EU input
    - All 0’s on both EU inputs
    - All 1’s on both EU inputs
    - Run all operations twice in a row (LDI, LDI, ADD, ADD etc.)
  + Check for ‘X’ and ‘Z’ inputs
  + Automate simulator commands through TCL/.do file
  + (Emulation compatible)
  + TLM Viewer in Questa
  + Section on tests
  + Section on sequences

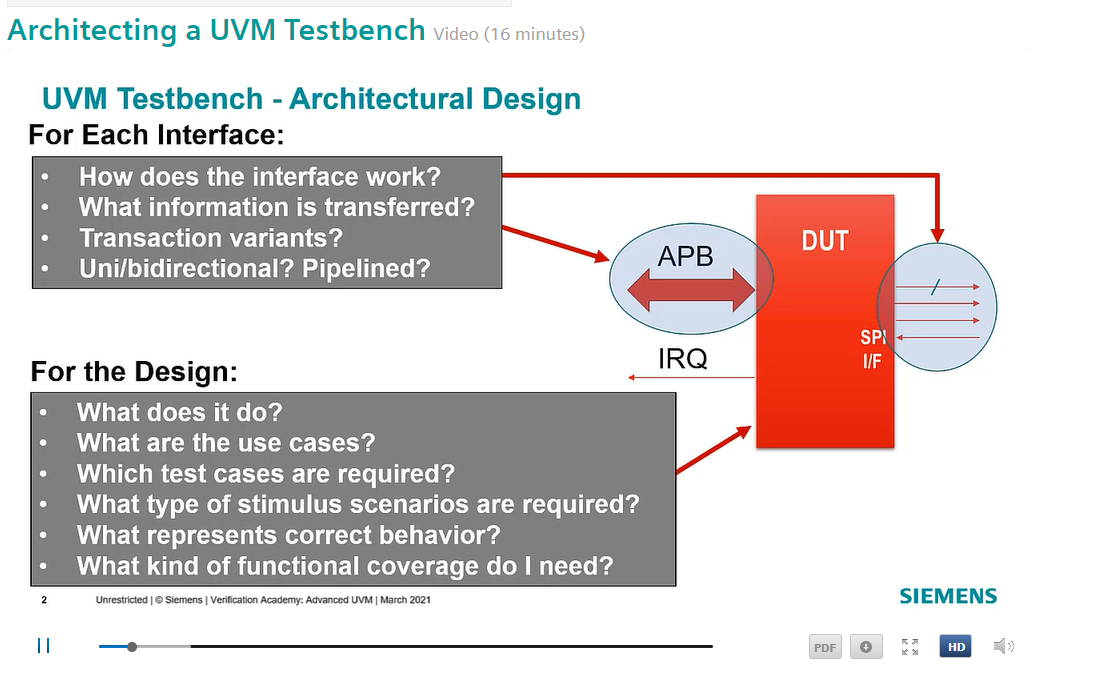


Figure 8 Questions to answer in this section (verificationacademy.com). Delete when section is finished.

# Micro Architecture

This chapter describes the micro-architecture of the processor. It describes the two main parts of it: the Datapath and the Control Unit. Verilog descriptions of each are given in the next chapter.

## Top level

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

### Data Path

//Add high level description. Describe signals.

A diagram of a data stream

Description automatically generated

Figure 9 Top Level Overview of the Datapath

A diagram of a computer system

Description automatically generated

Figure 10 Detailed Overview of the Datapath

#### ALU and Shifter: Execution Unit (EU)

//Add description.? Add opcode “address” for each EU: 0xxx for arithmetic, 10xx for logic, and 11xx for shifter. Refer to 1.3.3. instructions for these.

A diagram of a algorithm

Description automatically generated

Figure 11 EU Architecture

#### Register File

//Add description.

A diagram of a computer

Description automatically generated

Figure 12 Architecture of the register file

### Control Unit

//Add description.

A diagram of a block diagram

Description automatically generated

Figure 13 Top Level View of the Control Unit

A diagram of a computer program

Description automatically generated

Figure 14 Micro architecture of the Control Unit, right hand side connects to the data path shown in Figure 10

#### Program Counter

A diagram of a program counter

Description automatically generated

Figure 15 High-level overview of the Program Counter

A diagram of a diagram

Description automatically generated

Figure 16 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 + off set 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**

In order to keep incrementing, branching, or even jumping, 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 increments state and increment the output. If reset and PL are still low the next clock cycle, the FSM will transition back to the first state and increment the output again. 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 1.4, the first option with dual states was chosen for simplicity.

#### Instruction Memory

As part of the effort to keep the scope manageable, the instruction memory will be combinational to facilitate single cycle execution (fetch, decode, execute). The instruction memory has a 16 bit address bus and contains 16 bit instructions which are formatted as described in section 1.3.2 on Instruction Formats.

A diagram of instruction and instructions

Description automatically generated

Figure 17 High-level overview of the Instruction Memory.

#### Instruction Decoder

As can be seen in the instruction overview in Table 2 on page 10, there are 6 major instruction types:

1. Using the Execution Unit with register(s)
2. Mem Read
3. Mem 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. The value of the control bits for the datapath can be deduced by comparing the instruction description in Table 2, on page 7, to the datapath overview in Figure 10 on page 15.

For example: for adding the contents of two registers, the major type is execution unit (EU) with registers. Looking at overview in Figure 10 on page 15 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 3.

Table 3 control bits for each instruction category

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **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 |
| ~~Branch on Negative~~ | ~~1~~ | ~~1~~ | ~~0~~ | ~~1~~ | ~~X~~ | ~~X~~ | ~~0~~ | ~~0~~ | ~~1~~ | ~~0~~ | ~~1~~ |
| Jump | 1 | 1 | 1 | X | X | X | 0 | 0 | 1 | 1 | X |

Simplifying this logic leads to the following logic for each control bit:

Table 4 Logic Formula's for Datapath Control Bits

|  |  |
| --- | --- |
| **Control Bit** | **Logic (instruction bits)** |
| MB | 15 |
| RW | \_\_  14 |
| MD | 13 |
| Op\_Select [3:0]  3:  2:  1:  0: | 12  11  10 \_\_  9 and 15 |
| PL | 15 AND 14 |
| JB | 13 |
| BC | 9 |
| Rd[2:0] | [8:6] |
| rsA[2:0] | [5:3] |
| rsB[2:0] | [2:0] |

A graphical representation of the decoder is shown below.

A diagram of a machine

Description automatically generated

Figure 18 Instruction Decoder Logic

# RTL Design

This chapter provides the Verilog descriptions 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.

//Check if github URL is present in all source files when done. Auto add through github terminal? “Sign files”?

## Top Level

### Datapath

//The datapath’s most important parts are the the Execution Unit and Register file. The EU does this, the Reg file does…

#### Top Layer

//Show picture of datapath top layer with reference. Add RTL for datapath top layer. Then build unit test and add to section 4.1.

##### Execution Unit

The RTL for the Execution unit can be divided into an Arithmetic, Logic, and Shifter section as was shown in section 2.1.1.1 ALU and Shifter: Execution Unit (EU). An overview is shown again below for reference.

A diagram of a algorithm

Description automatically generated

Figure 19 Overview of the architecture of the Execution Unit as shown in2.1.1.1 on page 15

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.

A screenshot of a computer program

Description automatically generated

Figure 20 Top level RTL of the Execution Unit

###### Arithmetic

The Verilog description of the arithmetic unit is shown below. It performs arithmetic operations based on the value of op\_select. The instructions are referenced as comments and were described in section 1.3.3 on page 10.

A screenshot of a computer program

Description automatically generated

Figure 21 RTL design of the arithmetic part of the execution unit.

###### 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 1.3.3 on page 10.

A computer screen shot of a program

Description automatically generated

Figure 22 RTL design of the logic part of the Execution Unit

###### 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 1.3.3 on page 10.

A computer screen shot of a program code

Description automatically generated

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

##### Register File

The register file contains 8, 16-bit registers that can be loaded synchronously.

###### Top Level

The top-level description of the register file was provided in section 2.1.1.2 on page 16. The diagram is shown again below for reference. A diagram of a computer

Description automatically generated

Figure 24 High level overview of the register file (as shown in section 2.1.1.2)

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

* Inputs and outputs as shown in the aforementioned diagram.
* 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.

A screenshot of a computer program

Description automatically generated

Figure 25 Verilog description of the register file: ports, internal signals, and generate block for the registers.

A screenshot of a computer program

Description automatically generated

Figure 26 Verilog description of the register file: output multiplexers.

###### Single Register

The RTL for an individual register is shown below.

A computer screen shot of a program

Description automatically generated

Figure 27 RTL for a single register.

### Control Unit

A screenshot of a computer program

Description automatically generated

Figure 28 Verilog description of the Control Unit's top layer.

#### Program Counter

A diagram of a program

Description automatically generated

Figure 29 Simplified State Diagram of the Program Counter as shown in 2.1.2.1.

A screenshot of a computer program

Description automatically generated

Figure 30 Verilog description of the Program Counter: ports, state encoding, and state transitions.

A screen shot of a computer

Description automatically generated

Figure 31 Verilog description of the Program Counter: Next State Determination.

A computer screen with white text

Description automatically generated

Figure 32 Verilog description of the Program Counter: State Outputs.

#### Instruction Memory

As described in section 2.1.2.2 the instruction memory will be read-only and combinational.

A diagram of instruction and instructions

Description automatically generated

A screenshot of a computer program

Description automatically generated

Figure 33 Verilog description of the instruction memory holding a sample program.

#### Instruction Decoder

As described in section 2.1.2.3 on page 19, the instruction decoder is combinational and provides the mapping between the instruction bits and the control bits. The Verilog implementation is shown below.

A diagram of a machine

Description automatically generated

Figure 34 The Instruction Decoder as shown in 2.1.2.3 (shown here for easy reference)

A computer screen shot of a program

Description automatically generated

Figure 35 Verilog description of the instruction decoder.

# Verification

This chapter describes the verification efforts performed for this project. Individual blocks were tested bottom up with simple Verilog testbenches up to the datapath and control unit level. Overal system level testing was done by use of the UVM and is described in section 4.2.

## Unit tests (Verilog)

As an initial quality check for the RTL, simple unit tests were done by creating testbenches in Verilog. System level tests using UVM are described in section 4.2.

### Datapath

The datapath contains the Execution Unit, Register File.

#### Top Level

The top level contains the EU and register file and 2 muxes (Mux B and Mux D, named after the reference architecture in [1]). As a first test at the top level, all 8 registers were loaded with a value matching their name, i.e. reg 0 contains 0, reg1 contains 1, etc. Simulation results and Verilog code of the testbench are shown below.

A screen shot of a computer

Description automatically generated

Figure 36 Test of the datapath top level, loading all registers.

A screenshot of a computer program

Description automatically generated

Figure 37 Verilog code for the datapath's top level testbench: internal signals and module instantiation.

A screenshot of a computer

Description automatically generated

Figure 38 Verilog code for the datapath's top level testbench: Initial signal values plus control values for loading all 8 registers.

With the registers loaded, most of the instructions described in section 1.3.3 can now be tested. A single test is performed for each instruction type, except for memory writes and jump/branch instructions. Initial testing indicates that the operations perform as expected. Testbench results and Verilog code is shown below.

A screen shot of a computer

Description automatically generated

Figure 39 Figure 25 Test of the datapath top level, instruction execution

A screenshot of a computer program

Description automatically generated

Figure 40 Verilog code for the datapath's top level testbench: instruction execution.

The task “eu\_op”, short for execution unit operation, is shown below.

A screen shot of a computer program

Description automatically generated

Figure 41 Verilog of the "eu\_op" task referenced above.

#### Execution Unit

##### Top Layer

The top-level unit test for the execution unit is shown below. The actual tests were performed bottom up, i.e. arithmetic, logic, and shifter were tested individually first, however these tests are described in the next sections. The top-level test performs each instruction for a fixed, “random”, value of input A and B. Output shows the unit works correctly for these inputs.

A screenshot of a computer program

Description automatically generated

Figure 42 Top level test bench for the Execution Unit

A screenshot of a computer program

Description automatically generated

Figure 43 Top level test bench for the Execution Unit (continued)

A screenshot of a computer

Description automatically generated

Figure 44 Simulation results for the Execution Unit (waveform)

A screenshot of a computer code

Description automatically generated

Figure 45 Simulation results for the Execution Unit (text)

###### Arithmetic Unit

The testbench and results for the individual test of the arithmetic unit are shown below. The unit performs within specifications of the design.

A computer screen shot of a program

Description automatically generated

Figure 46 RTL of the unit test bench for the Arithmetic Unit.

A screenshot of a computer program

Description automatically generated

Figure 47 RTL of the unit test bench for the Arithmetic Unit (continued).

A screenshot of a computer

Description automatically generated

Figure 48 Simulation results for the Arithmetic Unit (waveform)

A table of numbers and symbols

Description automatically generated with medium confidence

Figure 49 Simulation results for the Arithmetic Unit (text)

###### Logic Unit

The logic unit testbench has a similar design compared to the one of the arithmetic unit shown above. The design passes all unit test cases.

A screenshot of a computer

Description automatically generated

Figure 50 Simulation results for the Logic Unit (waveform)

A screenshot of a computer code

Description automatically generated

Figure 51 Simulation results for the Logic Unit (text)

A screenshot of a computer program

Description automatically generated

Figure 52 Test cases for the logic testbench (Verilog)

###### Shifter

The testbench approach for the shifter is similar to the one taken for the arithmetic and logic parts. All testcases passed. The results are shown below.

A screenshot of a computer

Description automatically generated

Figure 53 Simulation results for the shifter (waveform)

A screenshot of a computer program

Description automatically generated

Figure 54 Simulation results for the shifter (text)

A screen shot of a computer program

Description automatically generated

Figure 55 Shifter testbench (Verilog)

#### 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)

##### Top Layer

The testbench instantiates a top layer as described in 3.2.1.2.1 on page 25. It contains a 50 MHz clock, frequency picked arbitrarily, a monitor, and test cases. The 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 simple for the unit test: write to all registers, read from all registers, read and write at the same time.

The Verilog description and results are shown below.

A computer screen with text on it

Description automatically generated

Figure 56 Verilog description of the testbench for the register file: IO, module instantiation, and monitor.

A computer screen shot of a program code

Description automatically generated

Figure 57 Verilog description of the testbench for the register file: test stimuli.

Test results are shown below. The complexity of this part is arguably greater than the Execution Unit’s top layer and because of that, the need for test automation and a more structural verification method, e.g. UVM, becomes evident. A stated before, system level testing will be done by using the UVM.

A screen shot of a computer

Description automatically generated

Figure 58 Test results for the top layer of the Register File (wave).

A blue and white lines

Description automatically generated with medium confidence

A blurry image of a building

Description automatically generated

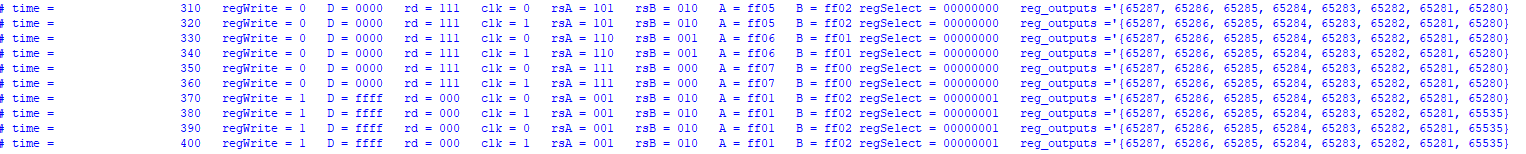


Figure 59 Test results for the top layer of the Register File (text).

###### Single Register

The test results for single register are shown below. The testbench design is similar to the top level but simpler. For the sake of brevity, it is omitted from this document.

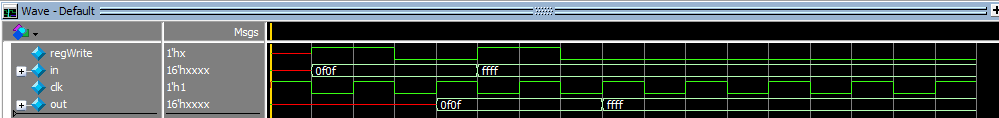


Figure 60 Testbench results for an individual register (wave)

A screenshot of a computer code

Description automatically generated

Figure 61 Testbench results for an individual register (text).

### Control Unit

The Control Unit’s top layer consists of three components: the Program Counter, PC, Instruction Memory, and the Instruction Decoder. All were tested separately and were then connected and test at the top level described in this section. Block level tests are described in the next sections.

A screen shot of a computer program

Description automatically generated

Figure 62 Verilog testbench for the control unit's top layer.

A screen shot of a computer

Description automatically generated

Figure 63 Testbench results of the Control Unit's top layer.

A screen shot of a computer

Description automatically generated

Figure 64 Testbench results of the Control Unit's top layer (continued).

#### Program Counter

The testcases and results for the Program Counter are shown below.

A screenshot of a computer program

Description automatically generated

Figure 65 Testcases for the Program Counter (Verilog testbench)

A screen shot of a computer

Description automatically generated

Figure 66 Simulation results for the Program Counter: part 1.

A screen shot of a computer

Description automatically generated

Figure 67 Simulation results for the Program Counter: part 2.

#### Instruction Memory

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

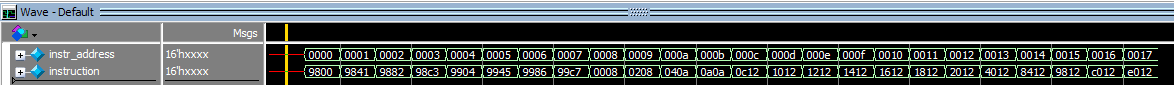


Figure 68 Simulation results for the Instruction Memory.

The Verilog testbench is simple:

A computer screen shot of a program

Description automatically generated

Figure 69 Verilog testbench for the Instruction Memory.

#### Instruction Decoder

To test the instruction decoder, a single major type of instruction, as described in 2.1.2.3 on page 19, was tested and the outputs were shown to be correct. The results and Verilog description of the testbench are shown below.

A screen shot of a computer

Description automatically generated

Figure 70 Simulation results for the Instruction Decoder.

A computer screen shot of a program

Description automatically generatedA screen shot of a computer program

Description automatically generated

Figure 71 Testbench for the Instruction Decoder

## Systel Level Test (UVM)

Include Verilog top level test (below)?

A screenshot of a computer

Description automatically generated

Figure 72 Top Level Testbench (Verilog) covering each instruction type.

### Verification Strategy

.

A diagram of a diagram

Description automatically generated

### Testbench Architecture Overview

Since the DUT only has a few connections, and a single interface, a block level architecture as shown below will be used for the testbench

### Interface

The testbench uses a single interface which is stored in the uvm\_config\_db. It contains two mod ports: one for the driver and one for the monitor.

Table 5 Interface Signal Overview

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Signal | Description | Origin | | Modport\_drv | | Modport\_mon | |
| Clk | Clock signal | | Top lvl input | | Output | | Input | |
| Reset | Reset | | Top lvl input | | Output | | Input | |
| Data\_in[15:0] | Data input from memory | | Top lvl Input | | Output | | Input | |
| Instruction[15:0] | Instruction to be executed | | Ctrl\_top.i\_mem | | N/A | | Input | |
| Instr\_addr[15:0] | Address for instr. register | | Ctrl\_top.pc | | N/A | | Input | |
| [0:7] reg\_outputs [15:0] | Register in reg file | | Dp\_top.rf\_reg\_top | | N/A | | Input | |
| Bus\_D[15:0] | Muxed: Execution Unit output or memory output (data\_in[15:0]) | | Dp\_top | | N/A | | Input | |
| Address\_out[15:0] | Address to PC / Memory (Bus A) | | Dp\_top | | N/A | | Input | |
| Data\_out[15:0] | Data to Memory (Bus D) | | Dp\_top | | N/A | | Input | |
| zero | ALU Status Bit | | Dp\_top.eu.arithmetic | | N/A | | Input | |
| Op\_select[3:0] | Selects operation to be performed for execution unit | | Ctrl\_top.instr\_dec | | N/A | | Input | |

A diagram of a computer

Description automatically generated

Figure 73 Block Diagram of the Interface to the Testbench.

### Test(s)

### Use Cases & Transactions

### Scoreboard

### Functional Coverage

### Transactions

### Score (scoreboard) (combine with functional coverage chapter?)

# Future improvements and lessons learned

## Potential improvements (outside current scope)

The architecture of this project was kept simple to make the scope manageable and compress the schedule of the project to one quarter of after-hours work. Its performance is too limited for practical applications. Low-cost microcontrollers are available that would beat its performance handsomely. The architecture is educational in nature and lacks features or capabilities such as:

* ISA Improvements:
  + Pick an actual ISA: RISC-V, ARMv8 etc.
* Micro-Architectural improvements:
* RAM for the instruction memory so different programs can be loaded.
* ALU: check for / handle overflow.
* ALU: handle signed numbers
* Cache memory
* Branch prediction
* More/advanced instructions (e.g. include a Floating Point Unit, FPU)
* Power management (sleep/awake/power levels)
* Interrupts
* Stack Pointer
* Pipelined architecture
* Super scalar architecture
* ADC/DAC, UART, I2C, SPI, CAN bus, Bluetooth etc.
* Dedicated HW support for AI (e.g., neural network accelerator, Multiply and Accumulators)
* Software/Firmware improvements:
  + Python script to facilitate an assembly kind of language that creates binary output for the instruction memory.

The inclusion, or absence, of these features would depend on the targeted market and product (embedded, mobile, high-performance compute, etc.) Also, this project features a single cycle design which limits the clock frequency and supported instructions. A pipelined design would speed up the throughput significantly.

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. This would also be an opportunity to investigate power, performance, and area requirements.

Nevertheless, the project’s goals were fully achieved:

* increased Verilog proficiency
* familiarity with more advanced features of the UVM
* enhanced knowledge of computer architecture.

## Lessons Learned

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

* Micro Architecture
* RTL Design
* Verification/Testbench Architecture
* Verification Engineering
* Program/Project Management

For future projects I will use the following lessons learned:

* Lorem
* Ipsum
* …

# 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. |

# Appendix I Control Unit – Detailed View

A diagram of a computer network

Description automatically generated

# Appendix II