Mini Project Report

on

GYMS-16 RISC-V Pipelined Processor

Submitted by

More Prasad (21BEC025)

R S Gokul Varun (21BEC034)

Sai Sathvik G B (21BEC042)

Yash Kumar (21BEC059)

Under the guidance of

Dr. Jagadish D N

Assistant Professor

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

INDIAN INSTITUTE OF INFORMATION TECHNOLOGY DHARWAD

25/11/2024

Certificate

This is to certify that the project, entitled **GYMS-16 RISC-V Pipelined Processor**, is a bonafide record of the Mini Project coursework presented by the students whose names are given below during the Academic Year 2024-25 in partial fulfillment of the requirements of the degree of Bachelor of Technology in Electronics and Communication Engineering.

Roll No	Names of Students
21BEC025	More Prasad
21BEC034	Gokul Varun
21BEC042	Sai Sathvik
21BEC059	Yash Kumar

Dr. Jagadish D N
(Project Supervisor)

Contents

List of Figures

1 About RISC

Computer Architecture introduced a simpler approach in how instructions are to be made.
Introduction to RISC:[2]

- Reduced Instruction Set Computer (RISC) is a CPU design philosophy that emphasizes simplicity and efficiency by using a smaller set of instructions and was pioneered in the 1980s, with notable contributions from IBM and UC Berkeley.
- RISC uses limited set of simple fixed-length instructions, often executed in single-digit order clock cycles.
- Memory operations are limited to load and store instructions, while arithmetic and logic operations work only on processor registers. RISC processors heavily utilise pipelining to execute multiple instructions simultaneously, enhancing throughput.
- It's designs often rely on compilers to generate efficient machine code, shifting complexity from hardware to software.

Advantages and Disadvantages of RISC:[2]

- The simplified instruction set enables faster execution, easier pipelining, low-power consumption makes it ideal for mobile and embedded systems.
- RISC architectures are easily scalable to support high-performance computing, thereby simplifies hardware, reducing cost and development time.
- But a problem is RISC requires more sophisticated compilers to optimise code simple instructions and often result in longer programs.

2 Why RISC?

This is comparison of RISC with other processor architectures, which are thriving in the market.
RISC vs CISC:[2]

- RISC has less no. of simplified yet fixed-length instructions, whereas CISC uses large no. of complex variable-length instructions.
- Due to this, RISC instructions typically execute in single-digit order clock cycles, while CISC instructions vary in execution times.
- For achieving this, RISC relies on registers for computations, whereas CISC can directly manipulate memory / use registers / accumulate into one entity.
- These qualities make RISC processors simpler, cheaper to design than CISC as well as popular.

RISC vs ARM:[2]

- ARM (Advanced RISC Machine) is specific implementation of RISC principles, optimised for low-power and high-efficiency, but RISC follows general-purpose design principles. It's inferred that ARM is a subset of RISC.
- ARM extends its functionality with fixed-length yet adaptable instructions for specific tasks. RISC is ued in academic and startup purposes whereas ARM dominates mobile markets.

RISC vs GPUs:

- RISC is designed for general-purpose computing, while GPUs are specialized for parallel processing tasks; so it's hard to compare them due to the application. GPUs are optimised for Single Instruction Multiple Data works and excel in massively parallel operations, due to which their efficiency varies.
- RISC processors are commonly used in embedded systems, mobile devices and servers, and GPUs are prominent in AI, ML, gaming and scientific computing.

3 GYMS-16 Specifications

- Total of 13 Arithmetic, Logical, Shift and Data movement Instructions.
- Instruction memory size of 0.5kB (8 bit address and 16 bit word)
- Data memory of same size; Harvard Architecture memory of total size 1 kB (0.5kB + 0.5kB)[1]
- General Purpose Register file is of size 32 Bytes (4 bit address and 16 bit word)
- Assembler written in Python for converting the Assembly-code into Binary/Hexadecimal output.

This figure shows how the processor instruction flows. The reverse arrows indicate that the read port data from register file and memory.

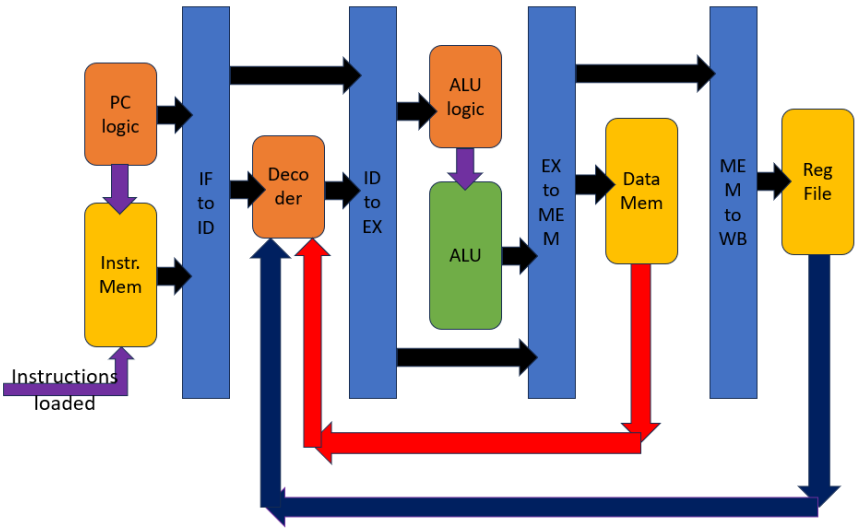

Figure 1. Design Flow

4 Instruction Set Architecture (ISA)

- The general format of GYMS-16 instruction is 4-bit opcode, followed by register addresses or memory addresses or conditions for easier ALU operand extraction.
- Arithmetic and Logical instructions except NOT are only-register based; while NOT involves condition to perform 0's or 1's complememt.[1]
- Shift instructions involve arithmetic and logical conditions along with the amount to be shifted as an immediate; Move instruction doesn't care the last 4 bits and just involves two 4-bit register addresses.

The following figures show how the instructions are made as written above.[1]

Figure 2. Arithmetic Operations

Shift Operations:

1. **SHL:** Shifts the bits of a register to the left by a specified number of positions, filling the empty bits with zeros.
   - *Logical left Shift*

1000	REGD(11-8)	REGS(7-4)	00	IMM(1-0)

   - *Arthimetic Left Shift*

1000	REGD(11-8)	REGS(7-4)	11	IMM(1-0)

2. **SHR:** Shifts the bits of a register to the right by a specified number of positions, filling the empty bits with zeros.
   - *Logical Right Shift*

1001	REGD(11-8)	REGS(7-4)	00	IMM(1-0)

   - *Arithmetic Right Shift*

1001	REGD(11-8)	REGS(7-4)	11	IMM(1-0)

Figure 3. Shift Operations

Logical Operations:

1. **AND:** Performs bitwise AND between two registers.
   - *FORMAT*

0100	REGD(11-8)	REGS1(7-4)	REGS2(3-0)

2. **OR:** Performs bitwise OR between two registers.
   - *FORMAT*

0101	REGD(11-8)	REGS1(7-4)	REGS2(3-0)

3. **NOT:** Performs bitwise NOT between a reggister and a value.
   - *For 1s complement*

0111	REGD(11-8)	REGS(7-4)	0000

   - *For 2s complement*

0111	REGD(11-8)	REGS(7-4)	1111

4. **XOR:** Performs bitwise XOR between two registers.
   - *FORMAT*

0110	REGD(11-8)	REGS1(7-4)	REGS2(3-0)

Figure 4. Logical Operations

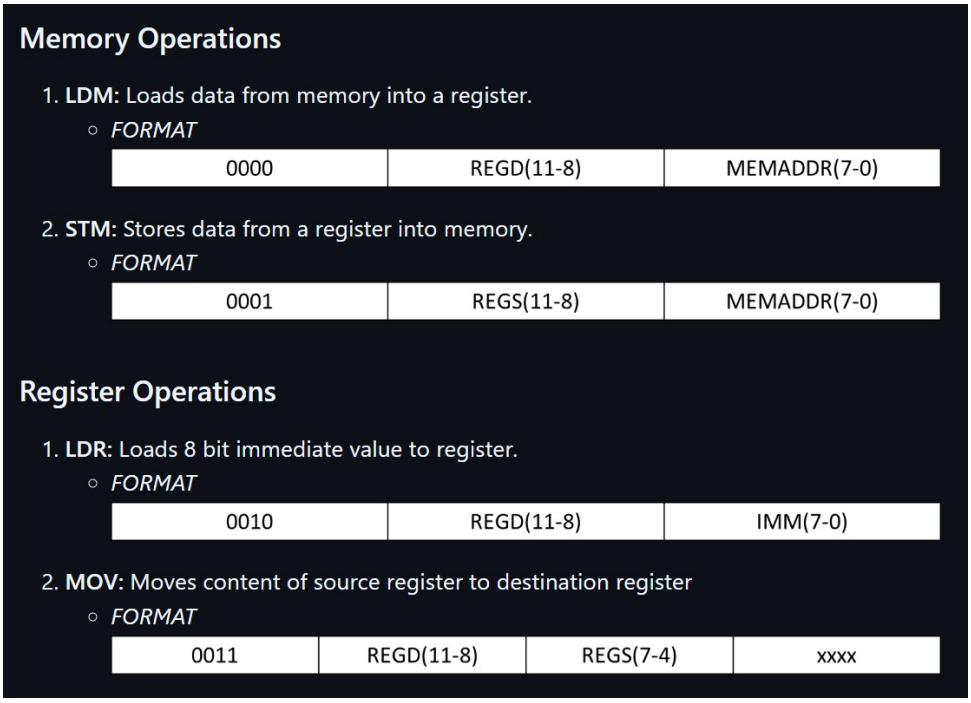

Figure 5. Memory and Register Operations

5 Pipelining

Pipelining in RISC Processors:[2]

- Pipelining is fundamental design principle in modern RISC architecture-processors, that enhances instruction throughput by overlapping the execution of multiple instructions.
- The concept is analogous to an assembly line in manufacturing, where different stages of a task are performed simultaneously.
- Pipelining is particularly effective due to the simplicity and uniformity of their instruction set, making it easier to divide instruction execution into discrete stages.
- In a non-pipelined processor, each instruction goes through all phases of execution sequentially. Pipelining addresses this inefficiency by splitting the instruction execution into stages that can operate concurrently.

Advantages and Challenges of Pipelining in RISC[2]

- By overlapping instruction stages, a RISC processor can complete one instruction per clock cycle, ensuring that all hardware resources are utilised efficiently.
- The fixed-length, uniform instructions of RISC simplify pipeline design. Pipelining minimises delays in executing consecutive instructions, enhancing performance.
- Increasing the number of pipeline stages can lead to higher throughput but also increases the complexity of hazard management and may lead to increased power consumption.
- Additionally, RISC's reliance on load/store operations can result in delays in the memory element.

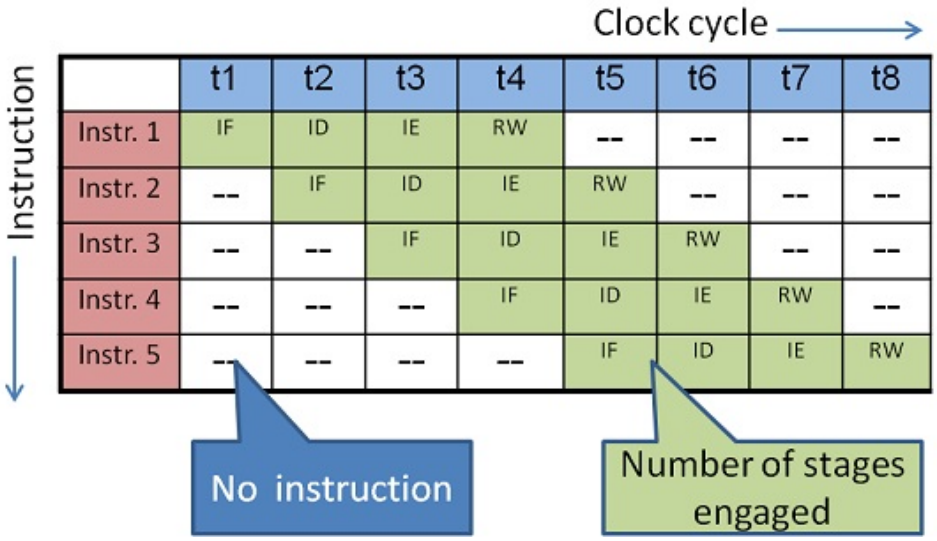

Figure 6. Picture of how pipelining works

Stages in Pipelining:[2]

- **Instruction Fetch (IF):** In this stage, the processor fetches the instruction from memory. The program counter (PC) is used to determine the address of the next instruction.
- **Instruction Decode (ID):** The fetched instruction is decoded to identify the operation to be performed. In this stage, the processor also reads the necessary registers or operands for the instruction.
- **Execution (EX):** The arithmetic or logical operation specified by the instruction is performed. For load/store instructions, the effective address is calculated.
- **Memory Access (MEM):**If the instruction involves memory (like load or store), this stage handles reading from or writing to memory.
- **Write Back (WB):** The result of the execution is written back to the register file, completing the instruction's life-cycle.

6 Hazards

Pipeline Hazards:[2] While pipelining improves performance, it also introduces certain hazards. These hazards occur when instructions in the pipeline interfere with one another, leading to stalls or incorrect results. There are three primary types of pipeline hazards, namely:

- **Structural Hazards:** These occur when two or more instructions simultaneously compete for the same hardware resource.
- **Control Hazards:** These arise from branch instructions, where the pipeline must decide which instruction to fetch next.
- **Data Hazards:** These occur when an instruction depends on the result of a previous instruction still in the pipeline. Data hazards are categorised into:
  - **Read After Write (RAW):** A subsequent instruction reads a register before a previous instruction writes.
  - **Write After Write (WAW):** Two instructions write to the same register in an overlapping manner, potentially overwriting data.
  - **Write After Read (WAR):** A later instruction writes to a register before an earlier instruction has finished reading.

Techniques to Mitigate Pipeline Hazards:

- **Forwarding or data bypassing:** It allows the pipeline to use the output of an instruction before it is formally written back to the register file.
- **Stalling:** The pipeline may insert no-op instructions to delay execution until it's resolved.
- **Branch Prediction:** The processor predicts the outcome of a branch instruction to continue fetching instructions without stalling.
- **Pipeline Interlocks:** Hardware mechanisms detect hazards and introduce stalls as necessary.
- **Speculative Execution:** Instructions following a branch are executed speculatively, assuming a predicted branch outcome.

7 Assembler

- An assembler is a key component in translating assembly language code into machine code for processor execution. Assembler design involves the processor's architecture and ISA.
- Writing in binary or hexadecimal is impractical for humans, so assembly language simplifies this process and assembler is the mediator between low-level language and processor.
- Since each processor has unique ISA, it ensures correct encoding for the processor, thereby giving low-level control of the hardware; useful in performance-critical applications.
- The assembler also checks for syntax errors, undefined symbols or invalid instructions and only allows correct code to enter the processor.

Phases of an Assembler:

- **Lexical Analysis:** This involves breaking down the assembly code into meaningful units or operands comprising adrresses, immediates and conditions.
- **Syntax Analysis:** The assembler checks if the tokens confirm to the ISA's rules, like correct number of operands and valid registers or immediate values; errors here are invalid instructions and wrong operand types.
- **Instruction Encoding:** Each assembly instruction is translated into its binary code based on the encoding scheme which consists of opcodes, registers and memory addresses.
- **Output Generation:** The assembler produces an output file, typically in binary, hex or object (machine code along with metadata) formats.

Types of Assemblers:

- Single-Pass Assembler resolves symbols and encodes instructions in a single pass; is faster but limited to simpler programs.
- Two-Pass Assembler builds the symbol table in first pass and encodes instructions using this table in second pass; it supports forward references.

Challenges in Assembler Design: Encoding common CISC architecture instructions requires sophisticated logic.

8 Simulation Results

The following pictures are the module-specific waveforms of the simulation; the waveforms were taken from the datapath so as to show how the data moves along the pipeline. The waveforms were generated from Icarus Verilog and GTKWave.

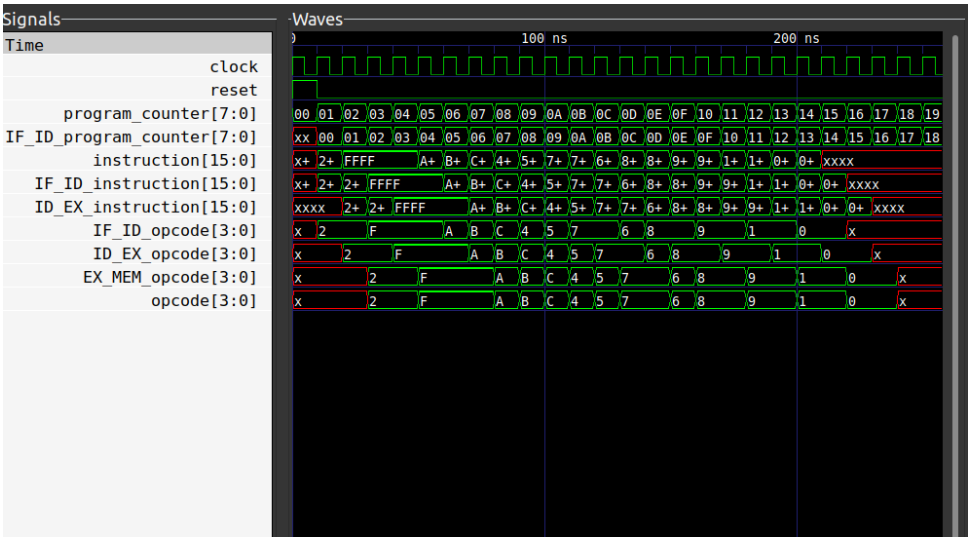

Figure 7. Pipelined movement of the instructions

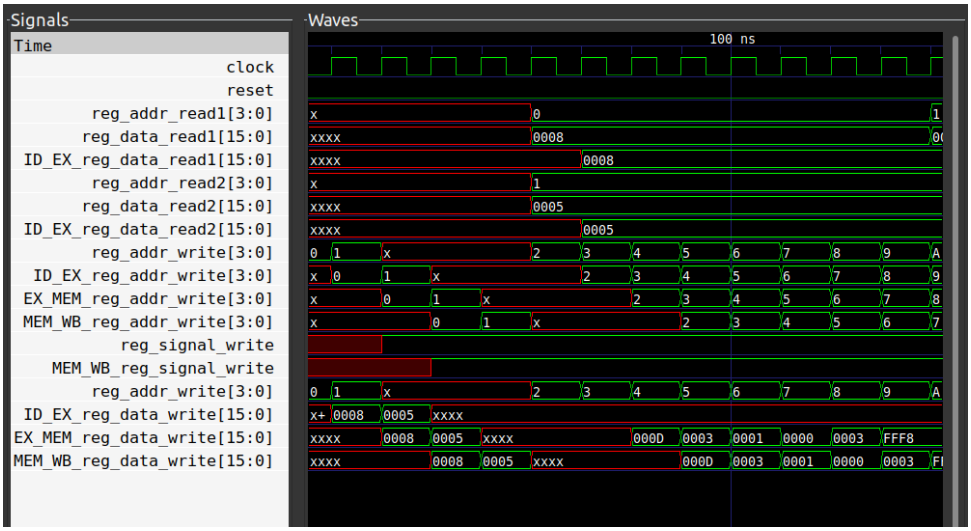

Figure 8. Register File read and write data with it's pipelined movement

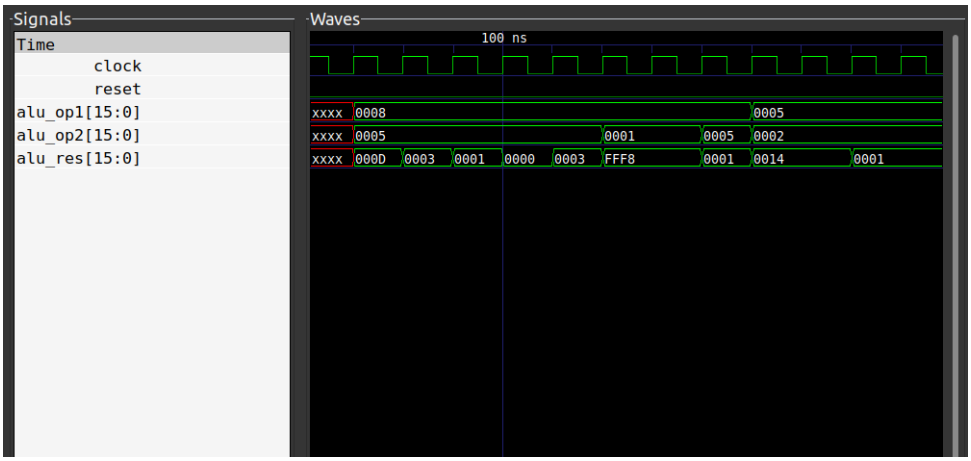

Figure 9. ALU workflow based on the test program (in next page)

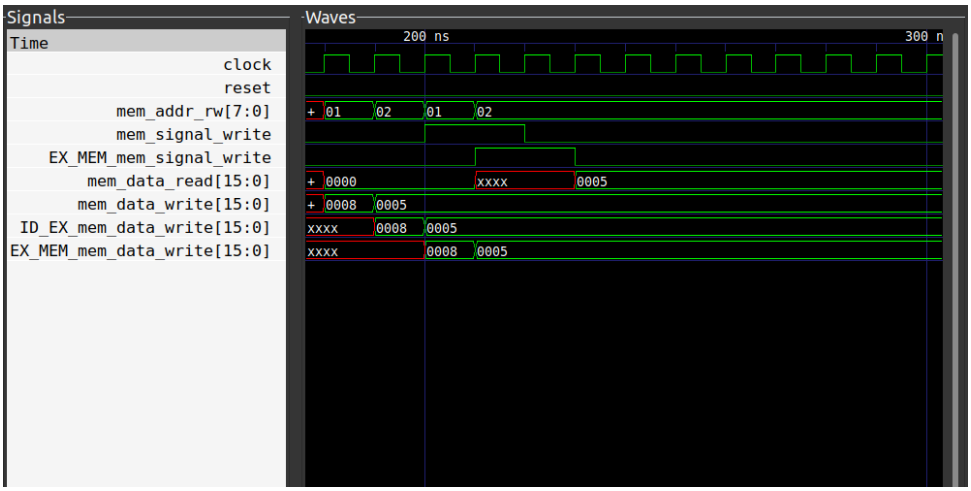

Figure 10. Data Memory read and write data with it's pipelined movement

9 Synthesised Designs

The following designs were synthesised from the Quartus Prime Lite Synthesiser tool.

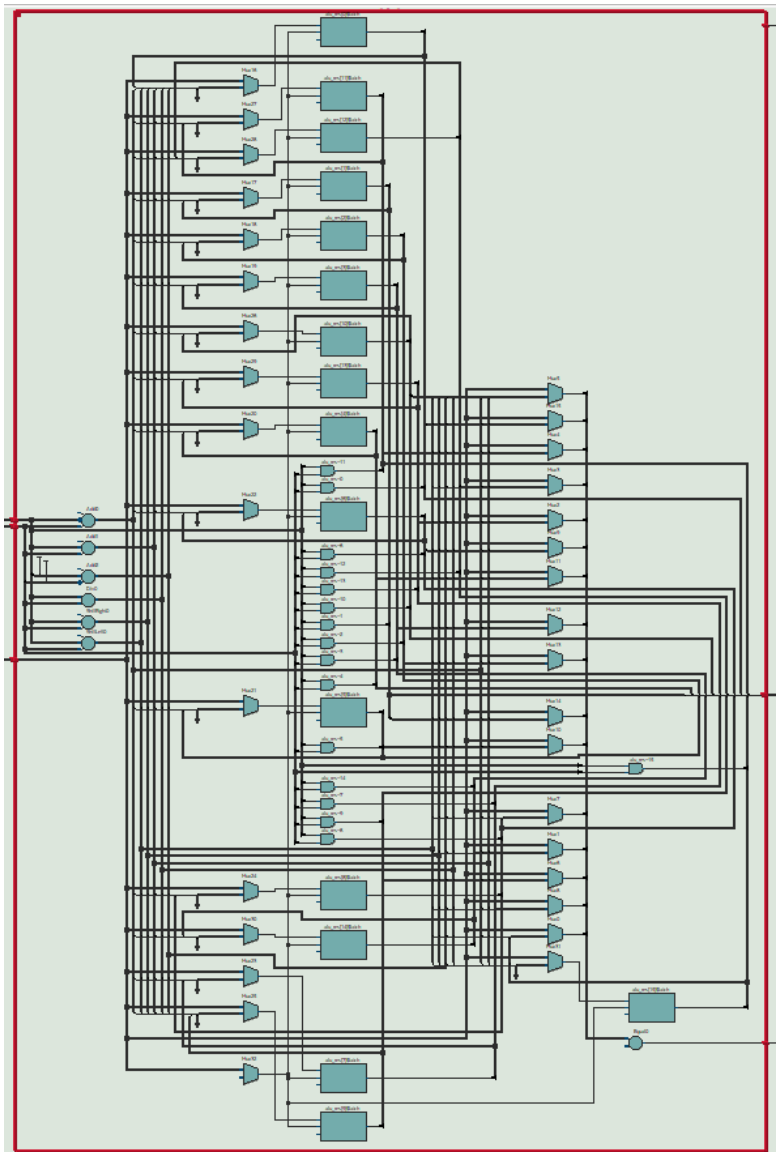

Figure 11. Arithmetic and Logical Unit (ALU)

```
LDR R0, 10   # Move immediate value 10 into register R0
LDR R1, 5    # Move immediate value 5 into register R1
NOP
NOP
NOP
ADD R2, R0, R1  # Add R0 and R1, store the result in R2 (R2 = R0 + R1)
SUB R3, R0, R1
DIV R4, R1, R0
STM R0, 00000001
LDM R5, 00000001
```

Figure 12. Assembly Program used for testing

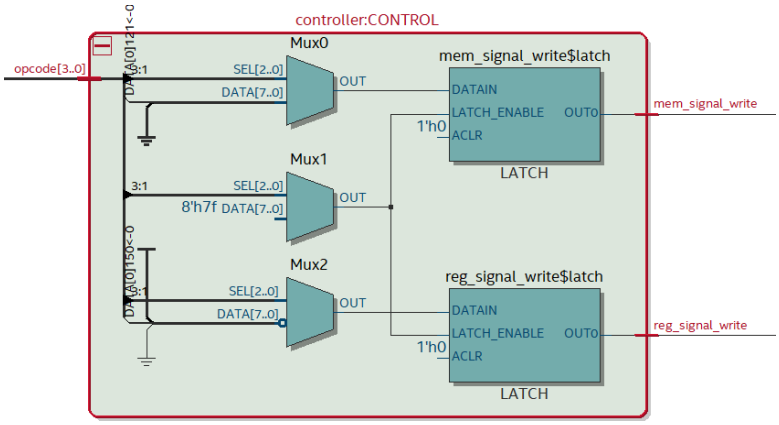

Figure 13. Controller

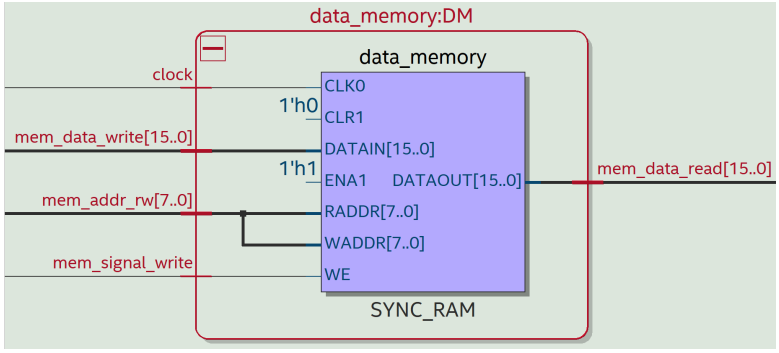

Figure 14. Data Memory

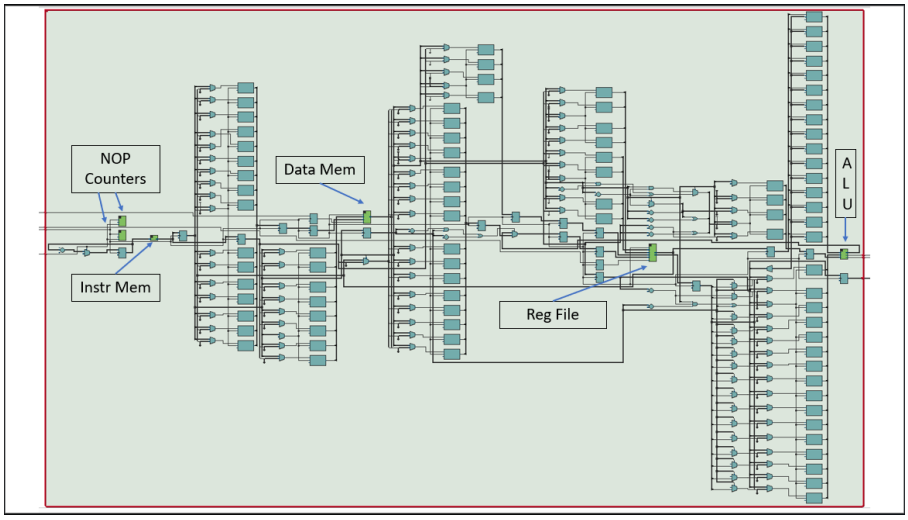

Figure 15. Datapath

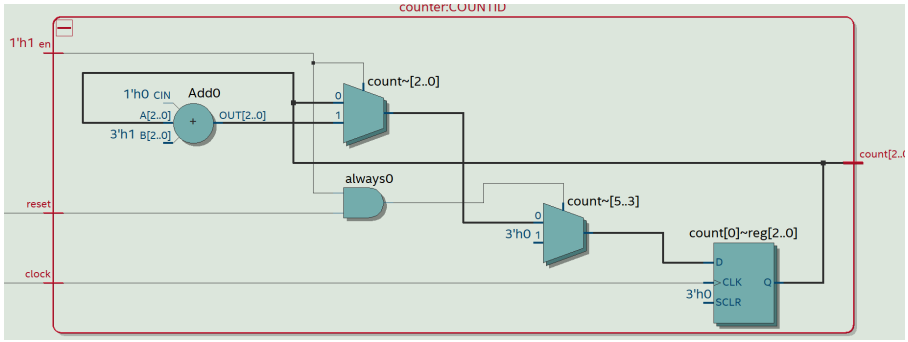

Figure 16. Inside an NOP counter

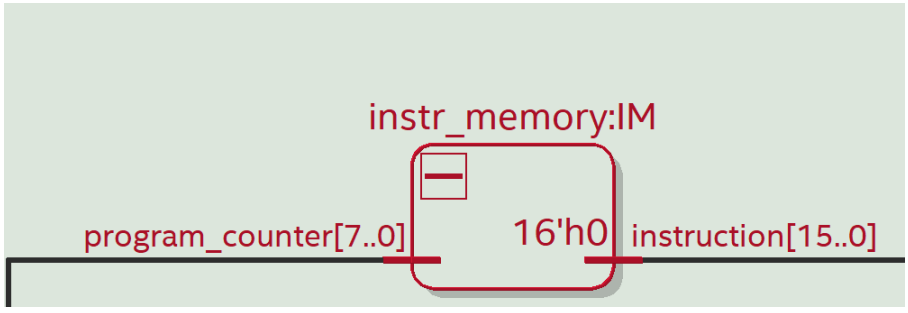

Figure 17. Instruction Memory

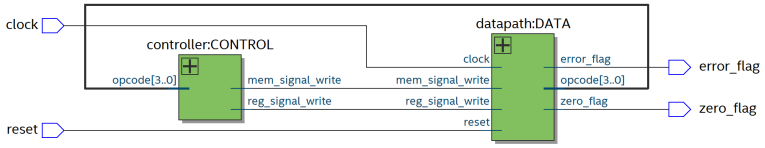

Figure 18. Processor Diagram

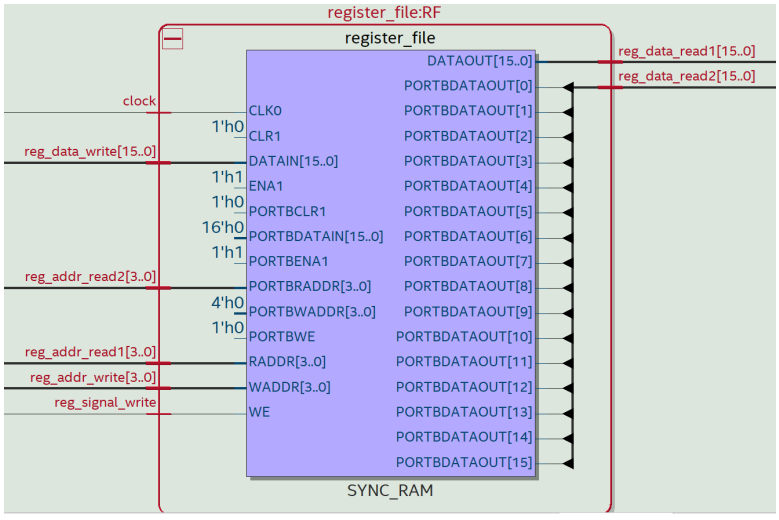

Figure 19. Register File

References

[1] FPGA4Student. Verilog code for 16-bit risc processor, 2017. URL `https://www.fpga4student.com/2017/04/verilog-code-for-16-bit-risc-processor.html`.

[2] Smruti Ranjan Sarangi. *Computer Organisation and Architecture*. McGraw Hill Education (India), New Delhi, 1st edition, 2015. ISBN 978-93-3290-183-4.