Baseline RISC-V Build

ELEC 5803

Jesse Levine (101185127)

1 Objective

The objective of this report and for phase 1 of the project is to reproduce the HLS RISC-V RV32I-based processor desribed in [1].

The baseline performance metrics are established (Area, Clock Frequency and Cycle Count) using a custom test program.

2 CPU Architecture

The processor is implemented as a multi-stage RISC-V core using HLS. it follows the standard Fetch-Decode-Execute cycle, interfacing with a unified memory block populated by the contents from a .txt file, *mem.txt*.

2.1 Instruction Pipeline

The core execution is divided into two primary loops: **Initilization Loop** and **Program Loop**.

In the Initialization Loop, the 32-bit entry Register File is cleared to ensure a deterministic start state.

The Program Loop operates in a continuous pipeline:

1. **Fetch**: The CPU asserts *mem_ce0* (Chip Enable) and reads the 32-bit instruction from memory and the current PC address.

2. **Decode**: The instruction is parsed to identify the Opcode, Source Registers and Destination Register

3. **Execute**: Arithmetic or logical operations are performed

4. **Write Back/Memory Access**: The result is written back to the Register File or, in the case of Store Word (SW) instruction, the CPU asserts *mem_weo* (Write Enable)

to save data back to RAM.

2.2 Memory Interface

The *mem.txt* file acts as the ROM for the simulation. At the start of the simulation, the testbench reads the hexadecimal machine code from *mem.txt* and pre-loads it into simulated RAM array.

The CPU treats this array as physical memory. If the program reaches the EBREAK instruction, the Program Loop terminates. asserting *ap_done* signal to indicate completion.

3 Methodology

To verify the design, we developed a ”Bare Metal” software flow to execute instructions directly on the core during C/RTL Co-Simulation.

The testing workflow proceeded as follows:

1. **Test Program Development:** A C program (`test.c`) was written to perform a fundamental arithmetic sequence (Store → Load → Add → Store). The source code is shown below:

Listing 1: test.c

```c
void main() {
    // 1. Setup: Create a pointer to address 1020 (0x3FC)
    // "volatile" prevents compiler optimization
    volatile int *ptr = (volatile int *)1020;

    // 2. Logic: Store 0, Load, Add 1, Store result
    *ptr = 0;           // sw x0, 0(x1)
    int val = *ptr;     // lw x2, 0(x1)
    val = val + 1;      // addi x2, x2, 1
    *ptr = val;         // sw x2, 0(x1)

    // 3. Stop: Force the processor to halt
    // Inserts the EBREAK opcode (0x00100073)
    __asm__ volatile ("ebreak");
}
```

2. **Compilation:** The C code was compiled into a RISC-V object file (`test.o`) using the GCC toolchain, targeting the base integer instruction set:

```
riscv64-unknown-elf-gcc -c -march=rv32i -mabi=ilp32 test.c -o test.o
```

3. **Hex Extraction:** The machine code was extracted and converted into the hexadecimal memory file (`mem.txt`) required by the testbench using `objcopy`:

```
riscv64-unknown-elf-objcopy -O binary -j .text test.o /dev/stdout | od -t x4 -An -w4 -v | sed 's/ //g' > mem.txt
```

   The resulting machine code is listed below. Each line represents one 32-bit instruction:

   Listing 2: Contents of mem.txt (Machine Code)

```
3fc00793   // li a5, 1020
0007a023   // sw zero, 0(a5)
0007a703   // lw a4, 0(a5)
00170713   // addi a4, a4, 1
00e7a023   // sw a4, 0(a5)
00100073   // ebreak
00008067   // ret (not executed)
```

4. **Simulation:** The resulting `mem.txt` was loaded into the Vitis HLS testbench. C/RTL Co-Simulation was performed to verify logical correctness via waveform analysis and to measure cycle latency.

5. **Implementation:** The design was exported to Vivado (Synthesis & Implementation) targeting the PYNQ-Z1 (xc7z020) to extract physical timing and area metrics.

4 CPU Verification

Functional correctness was verified via C/RTL Co-Simulation waveform analysis. The trace provided in Figure 1 captures the complete lifecycle of the test program execution, from instruction fetch to final data write-back.

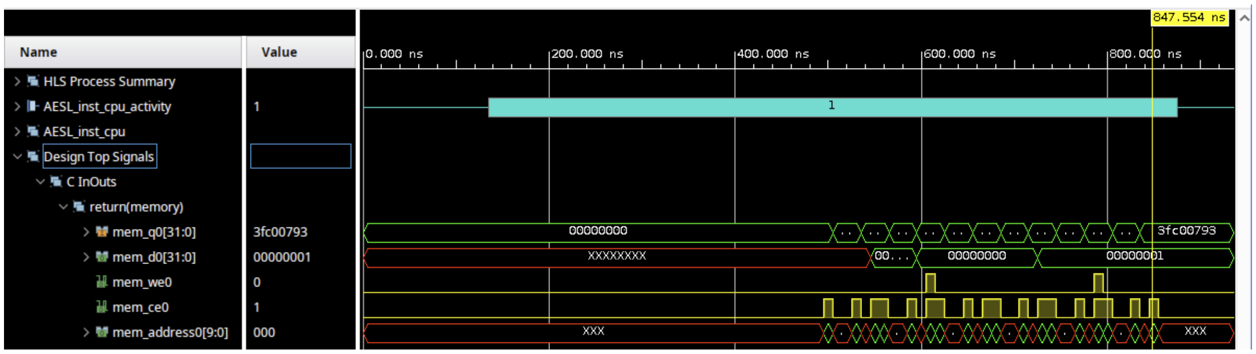

Figure 1: C/RTL Co-Simulation Waveform showing memory interface signals during execution. The markers highlight the instruction fetch and the final result write-back.

Observations

- **Program Counter Reset (`mem_q0`):** At the conclusion of the execution trace, the input data bus `mem_q0` stabilizes at `0x3fc00793`. This value matches the machine code for the first instruction at address `0x00` (`li a5, 1020`). This behavior confirms that upon completing the program and asserting `ap_done`, the core's Program Counter correctly resets to zero, causing the Fetch stage to speculative retrieve the entry-point instruction in preparation for the next run.

- **Data Write Events (`mem_we0` & `mem_d0`):** The Write Enable signal (`mem_we0`) asserts High exactly twice during the simulation window, corresponding to the two Store operations in the source code:

  1. **Initialization:** During the first pulse, the output bus `mem_d0` drives `0x00000000`, validating the execution of `*ptr = 0`.
  2. **Result Write-Back:** During the second pulse, `mem_d0` drives `0x00000001`. This confirms that the ALU successfully performed the addition $(0 + 1)$ and the processor correctly committed the result to memory.

- **Execution Cycles (`mem_ce0`):** The Chip Enable signal (`mem_ce0`) toggles Low intermittently throughout the trace. This behavior reflects the multi-cycle architecture of the core; memory access is disabled during internal Decode and Execute stages to maintain synchronous timing and reduce power consumption.

The expected C logic and the observed RTL signals confirm the logical correctness of the generated core.

5 Results & Metrics

5.1 Area Utilization

Table 1: Post-Implementation Area Utilization

Resource	**Count**	**Percentage**
LUT	1246	∼2.34%
FF	480	∼0.45%
BRAM	2	∼0.71%

5.2 Performance Metrics

Table 2: Performance Metrics Results

Metric	**Value**	**Source of Measurement**
Max Clock Frequency	130 MHz	Calculated from Post-Implementation critical path (7.692 ns).
Cycle Count (Total)	73 Cycles	Measured from `ap_start` to `ap_done` in Co-Simulation waveform.
CPI (Steady State)	6.33 CPI	Derived from the program execution loop (38 cycles / 6 instructions).

5.2.1 Metric Derivation

The cycle counts reported in Table 2 were derived directly from the C/RTL Co-Simulation waveforms by measuring the latency between the handshake signals. The clock period for the simulation was set to 10 ns.

Total Cycle Count (73 Cycles)

The total execution time was measured from the assertion of the top-level `ap_start` signal to the assertion of `ap_done`. As shown in Figure 2, the processor starts execution at 135 ns and completes at 865 ns.

$$\text{Total Latency} = \frac{865\,\text{ns} - 135\,\text{ns}}{10\,\text{ns/cycle}} = \frac{730\,\text{ns}}{10} = \mathbf{73}\ \text{Cycles}$$

This duration includes the initialization overhead (register file clearing) and the execution of the test program.

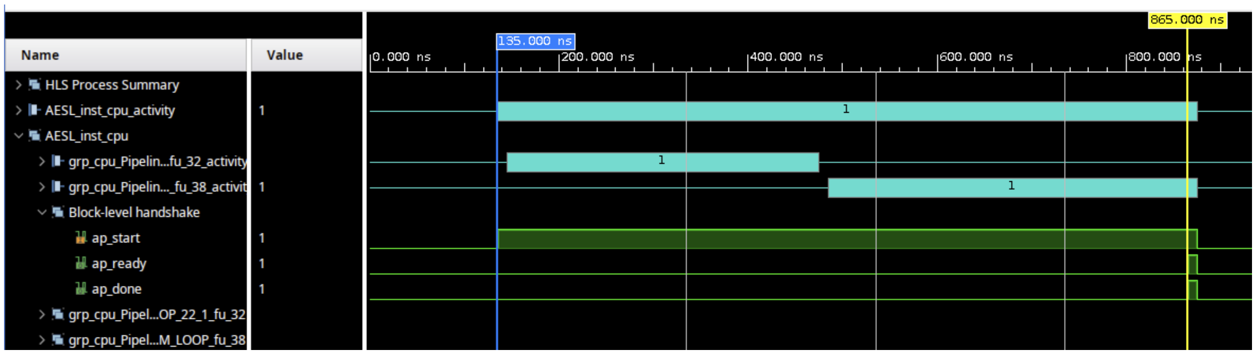

Figure 2: Measurement of Total Cycle Count. The cursors mark the start (135 ns) and end (865 ns) of the top-level execution.

Steady State CPI (6.33)

To calculate the Cycles Per Instruction (CPI) of the core logic, the `PROGRAM_LOOP` sub-block was isolated, which executes the instructions in `mem.txt`. As shown in Figure 3, this specific loop begins at 485 ns and concludes at 865 ns.

$$\text{Loop Latency} = \frac{865\,\text{ns} - 485\,\text{ns}}{10\,\text{ns/cycle}} = \mathbf{38}\ \text{Cycles}$$

The test program consists of 6 instructions. The CPI is calculated as:

$$\text{CPI} = \frac{38 \text{ Cycles}}{6 \text{ Instructions}} \approx \mathbf{6.33}$$

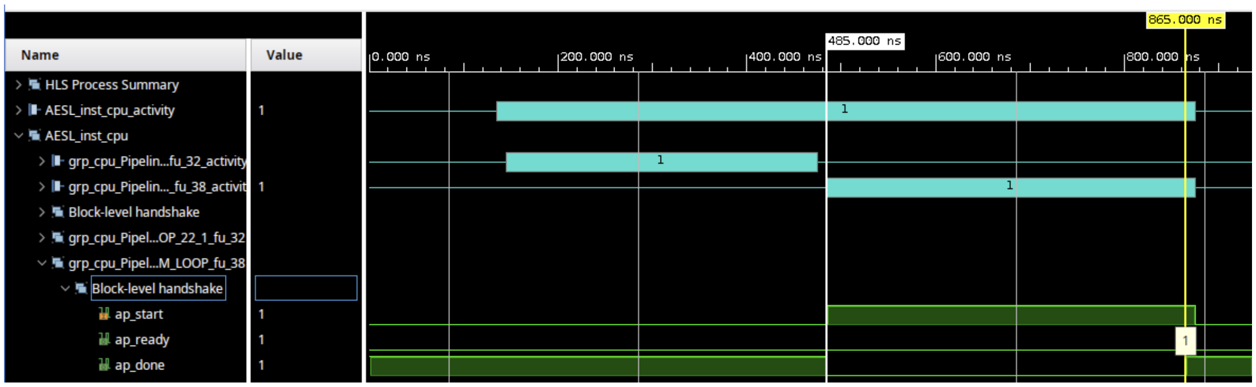

Figure 3: Measurement of Program Loop Latency. The cursors isolate the execution of the 6 test instructions, excluding initialization overhead.

6 AI Utilization

Gemini was used as a reference to connect the baseline architecture of the reference paper to both the HLS C++ implementation and the synthesis/implementation reports. I asked for a full-breakdown of the`riscv32i.cc`, source code, Gemini went through the code, line-by-line, confirming the `while(true)` structure as the continuous Fetch–Decode–Execute machine, mapping nested `switch(opcode)` / `switch(funcx)` logic to the control unit and ALU resource selection, and explaining how HLS directives (e.g., `#pragma HLS RESOURCE`) force `mem[]` into a single-port BRAM-style interface rather than a generic array.

It also interpreted HLS-specific syntax (`ap_int`/`ap_uint`, bit slicing such as `insn(y,x)`, immediate reconstruction for `immB`/`immS`, and byte write strobes via `wstrb`) so the decode/execute datapath was understandable at the bit level.

I then used Gemini to explain simulation and waveform behaviors; explicitly queried why `mem_q0` returned to `0x3fc00793` after program completion (auto-restart via `ap_start`/`ap_done` and Program Counter reset), what the toggling of `mem_ce0` implied about multi-cycle latency, and why `mem_we0` pulsed when it did (initialization writes versus final result commit).

For toolchain setup, I asked how to generate a bare-metal test program and produce the `mem.txt` hexadecimal format, the AI provided a minimal `test.c` pattern (volatile pointers plus an `ebreak` halt) and the exact `riscv64-unknown-elf-gcc / objcopy` steps to strip ELF overhead and emit RAM-initialization words.

Finally, AI was used to interpret synthesis outputs and quantify baseline performance: It showed how to isolate the steady-state `PROGRAM_LOOP` from initialization overhead, guided cursor placement to compute steady-state CPI (6.33), and derived maximum clock frequency (≈ 130 MHz) from the critical-path delay (7.692 ns), while also helping validate correctness by correlating observable interface activity (e.g., `mem_we0` assertions and the final write of `1` to address `1020`) directly back to the C store operations and intended program behavior.

Prompt Disclaimer

The above section 6 was written fully with AI using the following prompt:

Summarize this entire chat with a focus of how AI tools were used to understand the baseline architecture and synthesis reports. I wrote the attached report and need to fill in the AI Utilization section. Focus on what I asked for understanding the paper and architecture and what I asked/probed into how to write my own test file using the different elf-gcc/objcopy.

References

[1] O. Toker, "A high-level synthesis approach for a risc-v rv32i-based system on chip and its fpga implementation," *Engineering Proceedings*, vol. 58, 2023, presented at the 10th International Electronic Conference on Sensors and Applications (ECSA-10). [Online]. Available: https://doi.org/10.3390/ecsa-10-16212