EE533 Lab6 Report

Team 5 Members:
Jeremy Cai (yujiecai@usc.edu, 5988192173)
Prathamesh Hirekodi (hirekodi@usc.edu, 5404695932)
Yanzhe Li (yanzheli@usc.edu, 6884027371)

Date: February 21, 2026

1. Introduction

The goal of Lab 6 is to extend our pipelined processor datapath to execute a practical subset of the ARM ISA sufficient to run assembly generated by an ARM C compiler, and to demonstrate correctness by running a sorting program (bubble sort in the handout) on NetFPGA. In addition, the lab requires hardware augmentation to support four hardware threads with accelerated context switching (multiple PCs and a multithreaded register file), enabling parallel execution of multiple programs with minimal overhead.

2. High-Level System Overview

2.1 NETFPGA TOOL VM SETUP (BUILD/RUN WORKFLOW SUMMARY)

We used the NetFPGA tool VM workflow to synthesize and run our design. At a high level, the workflow is:

1. build the NetFPGA project to generate a bitfile,
2. program the FPGA,
3. use the host transaction/MMIO path to load IMEM/DMEM contents,
4. start the system, and
5. read back DMEM and debug state to validate correctness.

2.2 MODULE ORGANIZATION

Our hardware is organized as a small SoC around the CPU core:

- **cpu.v**: single-thread 5-stage pipeline (IF/ID/EX/MEM/WB) with ARM-style decode, conditional execution, forwarding/hazard handling, barrel shifter, multiply/MAC, and a multi-cycle block-transfer/swap engine.
- **cpu_mt.v**: quad-threaded 5-stage pipeline extending the base CPU architecture with zero-overhead hardware context switching to support four concurrent network processing threads.
- **soc.v**: MMIO-controlled wrapper around CPU + instruction/data memories, providing a simple request/response interface to program IMEM/DMEM and start/stop execution.
- **soc_driver.v**: transaction driver with FIFO buffering and transaction-quality monitoring (timeouts, counters), bridging “user transactions” into the SoC MMIO channel.
- **ila.v**: lightweight debug/control block providing clock gating, run/step control, and live internal probe readout.
- **top.v**: top-level integration of driver + SoC + ILA control.

Integration note (important for clarity):

- The SoC wrapper (soc.v) instantiates the single-thread core (cpu.v) in the default build.
- The multithreaded core (cpu_mt.v) is fully implemented and verified using a dedicated multithreaded testbench flow

(see Section 11), and can be integrated into the SoC by swapping the instantiated core module and aligning the memory interface expectations. *(We include both implementations in this submission to clearly demonstrate Part 1 and Part 2.)*

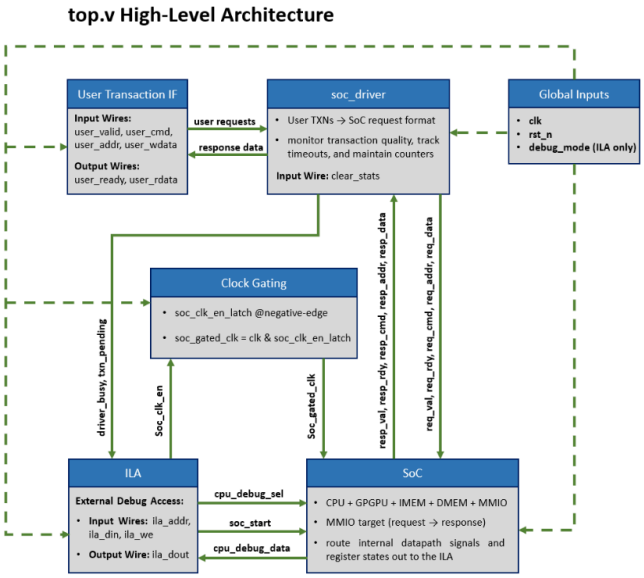


Figure 1. high-level design of top.v

3. ISA Compatibility Strategy and Supported Instruction Subset

3.1 ARM ENCODING APPROACH

We chose to implement decoding that follows ARM-style instruction formats (rather than inventing a custom ISA). The control unit (cu.v) extracts canonical ARM fields such as:

- cond = instr[31:28] (condition code)
- opclass = instr[27:25] (major instruction class)
- opcode = instr[24:21] (DP opcode)
- register fields Rn/Rd/Rs/Rm
- immediate rotate and offset fields for DP/SDT/branch

This allows us to directly use ARM GNU toolchain outputs (machine code) as IMEM contents, minimizing custom assembler complexity.

3.2 SUPPORTED INSTRUCTIONS

Our current RTL includes decoding and execution support for the following categories:

1. Data Processing (DP)

- Register DP and immediate DP forms.
- ALU ops implemented: AND/EOR/SUB/RSB/ADD/ADC/SBC/RSC/TST/TEQ/CMP/CMN/ORR/MOV/MVN.
- Conditional flag updates through CPSR (N/Z/C/V).

2. Barrel Shifter

- LSL/LSR/ASR/ROR with ARM special cases for immediate shift of 0 (e.g., LSR #0 ≡ LSR #32, ROR #0 ≡ RRX).

3. Single Data Transfer (SDT) + Halfword Data Transfer (HDT)

- Load/store with pre/post indexing and up/down address modes.
- Byte/halfword/word access size and signed/unsigned load extension.
- Base writeback (!P or W) via a secondary writeback port.

4. Branch

- B / BL: target computed with ARM semantics (PC-relative, using PC+8 convention).
- BX: register indirect branch/exchange (used for returns like bx lr).

5. Multiply / Multiply-Accumulate

- MUL/MLA and long multiply variants producing RdLo/RdHi (two write ports).

6. Multi-cycle instructions via BDTU

- LDM/STM (block data transfer) and SWP/SWPB handled by bdtu.v (multi-cycle FSM) while stalling the pipeline.

This superset is designed to cover what appears in compiler-generated sorting code (e.g., ldr/str, cmp, conditional branches, shifts, and ldmia/stmia patterns). The handout's bubble-sort assembly example includes these patterns.

4. Microarchitecture: 5-Stage Pipeline

Our CPU implements a classic 5-stage pipeline (IF/ID/EX/MEM/WB) based on the reference pipelined datapath described in the lab handout.

4.1 IF (INSTRUCTION FETCH)

- PC updates every cycle unless stalled by the hazard unit.
- Next PC is either PC+4 or a resolved branch target from EX stage.
- Because our instruction memory is synchronous, the design includes an "instruction hold" mechanism to preserve a fetched instruction during decode stalls.

4.2 ID (DECODE + REGISTER READ + CONDITION CHECK)

- cond_eval.v evaluates cond against CPSR flags (N/Z/C/V).
- cu.v decodes the instruction and generates control signals: operand source selects, immediate generation, memory controls, branch/mul flags, writeback routing, and register usage flags for hazards/forwarding.
- Register file reads are performed in ID and latched into ID/EX.

ARM PC semantic note: When R15 is read as an operand, we supply PC+8 (implemented as pc_plus4_id + 4) to match ARM behavior.

4.3 EX (EXECUTE)

1. Forwarding unit (fu.v) selects the most recent producer values for: Rn, Rm, Rs, and store-data (Rd) sources.
2. barrel_shifter.v shifts Rm if required (DP register shifts and register-offset addressing forms).
3. alu.v computes result and flags; flags are updated when S is set and the condition is met.

4. mac.v supports multiply/multiply-accumulate paths.
5. Branch target is computed in EX:

- For B/BL: target = PC+8 + imm32
- For BX: target = Rm

6. Branch decision is resolved in EX and triggers pipeline flush of younger stages.

4.4 MEM (DATA MEMORY / MULTI-CYCLE ARBITRATION)

1. For normal SDT/HDT, the CPU drives: d_mem_addr, d_mem_wdata, d_mem_wen, d_mem_size.
2. For multi-cycle BDT/SWP:

- bdtu.v takes priority over the memory bus and register write ports.
- While BDTU is busy, the hazard unit stalls IF/ID/EX/MEM to “freeze” pipeline state.

4.5 WB (WRITEBACK)

Writeback uses a mux (wb_sel) supporting:

- ALU result
- memory load data (with sign/zero extension for byte/halfword)
- link value (PC+4) for BL
- CPSR for MRS
- multiply result (RdLo), plus a secondary port for RdHi or base writeback.

5. Register File Design and Dual Writeback

To support ARM-style addressing writeback and long multiply results, our register file provides:

- 4 read ports (Rn, Rm, Rs, Rd/store/accumulate)
- 2 write ports (primary and secondary destination)
- combinational reads with simple same-cycle forwarding
- synchronous writes on clock edge

The CPU arbitrates write ports when the multi-cycle BDTU is active (BDTU writes have priority; spare capacity may be used by the pipeline WB path).

6. Hazard Detection and Forwarding

6.1 FORWARDING (FU.V)

We forward operands from multiple sources with a priority scheme:

1. EX/MEM port1 (youngest ALU result)
2. EX/MEM port2 (youngest “secondary” result: base writeback or RdHi)
3. MEM/WB port1

4. MEM/WB port2
5. BDTU port1
6. BDTU port2

This is necessary for ARM patterns such as:

- post-index or writeback loads followed by dependent uses of the updated base;
- long multiply producing RdHi/RdLo in one instruction;
- stores that need the latest store-data value.

6.2 HAZARD DETECTION (HDU.V)

We implement:

- Load-use hazard stall: if the instruction in ID depends on a load result in EX, we stall IF/ID and insert a bubble into EX for one cycle.
- Branch flush: when a branch is taken in EX, we flush IF/ID and ID/EX (wrong-path instructions).
- Multi-cycle stall: when BDTU is busy, we freeze the pipeline upstream to give BDTU exclusive control of regfile write ports and memory interface.

7. Multi-Cycle Support: Block Data Transfer and SWP (bdtu.v)

Compiler-generated ARM code for sorting can include block copies using ldmia/stmia (e.g., initializing stack frame arrays). The lab handout's example assembly contains these sequences.

To support these efficiently, we implemented bdtu.v, a finite-state machine that:

- iterates through a register list (LDM/STM) one register per cycle;
- handles synchronous memory read timing (pipeline "drain" state for the last load);
- optionally writes back the base register (W bit);
- implements SWP/SWPB as a read-then-write sequence with proper byte/word behavior;

During BDTU operation, the core asserts busy, and the hazard unit stalls the pipeline until the multi-cycle sequence completes.

8. MEMORY SYSTEM AND MMIO PROGRAMMING INTERFACE (SOC.V / SOC_DRIVER.V)

Lab 6 requires that we can load binary code into the processor and demonstrate correctness on NetFPGA. We provide an MMIO request/response channel in soc.v to:

- write/read instruction memory (IMEM);
- write/read data memory (DMEM);
- start execution via a control region. A simple region decode is used (high address bits select IMEM/CTRL/DMEM). While the CPU is active, host MMIO is blocked to avoid memory contention.

A simple region decode is used (high address bits select IMEM/CTRL/DMEM). While the CPU is active, host MMIO is blocked to avoid memory contention.

8.1 MMIO ADDRESS REGIONS (AS IMPLEMENTED)

We use req_addr[31:30] to select the target region:

- 00 → **IMEM** (instruction memory programming)
- 01 → **CTRL** (start/active status)
- 10 → **DMEM** (data memory programming)
- 11 → reserved

In our SoC, the CPU uses **byte addressing internally**, but the test memories are word-indexed; therefore the SoC converts CPU byte addresses to word indices using >> 2 for both IMEM and DMEM CPU paths.

8.2 HANDSHAKE AND RUN CONTROL

The SoC uses a valid/ready handshake (req_val/req_rdy, resp_val/resp_rdy). While the CPU is running (system_active=1), we deassert req_rdy to prevent host access conflicts and X-propagation. The CPU starts via a CTRL write (latching cpu_active) and clears when cpu_done is asserted.

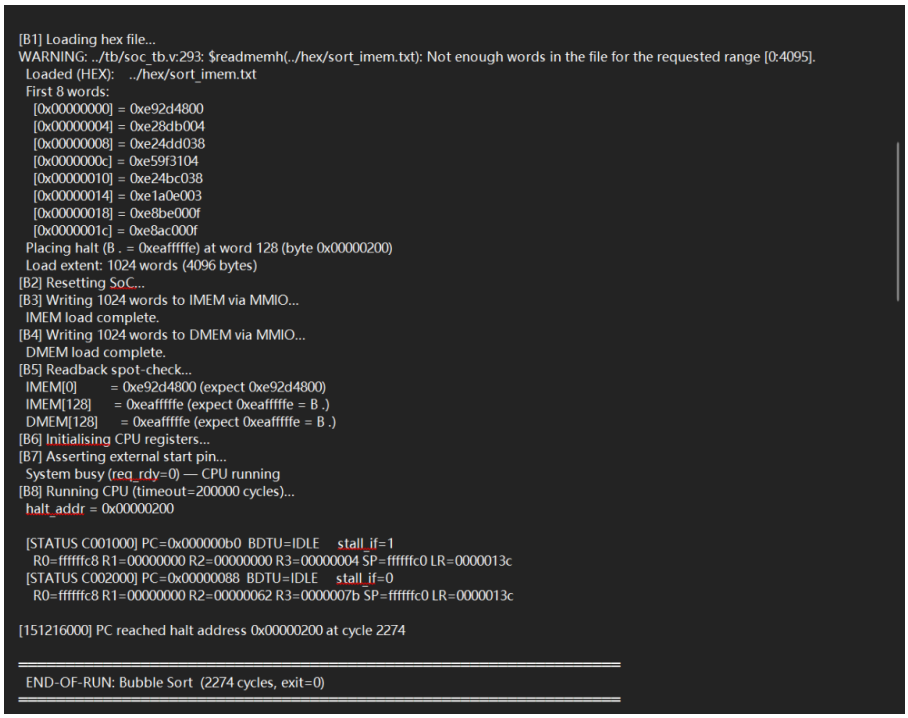


Figure 2. SoC MMIO Request/Response Flow

9. SOFTWARE FLOW: C → ARM ASSEMBLY → MACHINE CODE → IMEM/DMEM LOAD

The handout requires using the ARM GNU cross-compiler to generate assembly and then converting the assembly to binary machine code suitable for the processor. Our workflow is:

9.1 AUTOMATED BUILD SCRIPT (ASSEMBLER.SH)

Instead of manually invoking each tool, we use assembler.sh to generate all required artifacts from a single C source file:

1. Generate assembly (.s)

- arm-none-eabi-gcc ... -S <file>.c -o <file>.s

2. Compile to object (.o)

- arm-none-eabi-gcc ... -c <file>.c -o <file>.o

3. Link to ELF with a minimal linker script

- The script writes a minimal linker script that places .text at 0x0000_0000 (matching our IMEM base).
- arm-none-eabi-ld -T <file>.ld <file>.o -o <file>.elf

4. Generate disassembly for verification

- arm-none-eabi-objdump -d <file>.elf > <file>_disasm.txt

5. Extract raw binary and convert to Verilog/IMEM hex

- arm-none-eabi-objcopy -O binary <file>.elf <file>.bin
- xxd -e -c 4 <file>.bin | awk '{print $2}' > <file>_imem.txt

The xxd -e -c 4 option ensures little-endian 32-bit words are written one-per-line, which matches our memory loader conventions and prevents word/byte swapping issues.

9.2 LOADING IMEM/DMEM

For the single-thread SoC flow:

- We load <program>_imem.txt into IMEM by writing 32-bit instruction words sequentially.
- We initialize DMEM with the unsorted array (and any stack/data structures required by the program).
- We start execution using the CTRL region, wait for cpu_done, and then read back DMEM for correctness.

For the multithreaded flow (Section 11):
We compile four independent programs (network_proc0~3.c) and load them into distinct IMEM base regions (e.g., 0x0000, 0x1000, 0x2000, 0x3000), as shown in our multithreaded debug trace. (Screenshots/transcripts of the above commands and MMIO sequences will be included in the final submission.)

10. Sorting Program and Correctness Demonstration

Per the lab objective, we execute a sorting program that rearranges N numbers in memory into sorted order. We generated ARM assembly for a bubble sort program (file sort.s) and optionally also tested a quicksort variant (quick_sort.s) to stress BL/BX and recursion.

Correctness checks:

1. Dump DMEM contents before execution (unsorted array).
2. Run the CPU until cpu_done is asserted.
3. Dump DMEM contents after execution and verify the array is sorted.
4. Capture internal signals (PC/instruction/ALU/memory ops/CPSR flag using the ILA probe selection path).

In our simulation log, the bubble sort test completes successfully and reports **PASS** along with the total cycle count.

```
#######################################
###  TOP — BUBBLE SORT INTEGRATION   ###
#######################################

VCD info: dumpfile top_sort_tb.vcd opened for output.
  [PASS] Driver ready after reset
  Loading hex: ../hex/sort_imem.txt
WARNING: ../tb/top_sort_tb.v:282: $readmemh(../hex/sort_imem.txt): Not enough words in the file for the requested range [0:4095].
  Extent: word 0..128 (129 words)
  First 4: e92d4800 e28db004 e24dd038 e59f3104
  Halt @ word 128 (byte 0x00000200) = 0xeaffffffe

  Writing 129 words to IMEM...
    IMEM 50 / 129
    IMEM 100 / 129
    IMEM done (129 words)
  Writing 129 words to DMEM...
    DMEM 50 / 129
    DMEM 100 / 129
    DMEM done (129 words)
  [PASS] Program loaded to IMEM and DMEM

  Spot-checking...
  [PASS] IMEM[0] matches hex
    IMEM[0] = 0xe92d4800 (expected 0xe92d4800)
  [PASS] DMEM[0] matches hex
  [PASS] IMEM halt word = B .
  [PASS] DMEM halt word = B .

  Init CPU regs (SP = 0x00000800, LR = 0x00000200)...

  ████ Starting CPU ████
  [PASS] cpu_run_level = 1 (CPU running)
  Polling PC (timeout = 5000 cycles)...
  CPU halted: PC = 0x00000200 (cycle 2400)
  [PASS] CPU halted after sort execution
  [PASS] cpu_run_level = 0 (CPU stopped)
  [PASS] MMIO available after CPU stop
```

Figure 3. IMEM content loaded

```
  Reading 768 DMEM words...
    200 / 768
    400 / 768
    600 / 768
    Readback complete

  Searching for sorted sequence in 768 DMEM words...
  Sorted array at DMEM word 497 (byte 0x000007c4):
    [0] = 0xfffffe39  (-455)
    [1] = 0xffffffc8  (-56)
    [2] = 0x00000000  (0)
    [3] = 0x00000002  (2)
    [4] = 0x0000000a  (10)
    [5] = 0x00000041  (65)
    [6] = 0x00000062  (98)
    [7] = 0x0000007b  (123)
    [8] = 0x0000007d  (125)
    [9] = 0x00000143  (323)
  [PASS] Sorted array found in DMEM

  Verifying ascending order...
  [PASS] All elements in ascending signed order
  Comparing against expected...
    [PASS] arr[0] = -455
    [PASS] arr[1] = -56
    [PASS] arr[2] = 0
    [PASS] arr[3] = 2
    [PASS] arr[4] = 10
    [PASS] arr[5] = 65
    [PASS] arr[6] = 98
    [PASS] arr[7] = 123
    [PASS] arr[8] = 125
    [PASS] arr[9] = 323
  [PASS] All elements match expected values

  Post-sort register dump:
    R0  = 0x00000000  (0)
    R1  = 0x00000000  (0)
    R2  = 0x0000007d  (125)
    R3  = 0x00000000  (0)
    R4  = 0x00000000  (0)
```

Figure 4: DMEM after sort

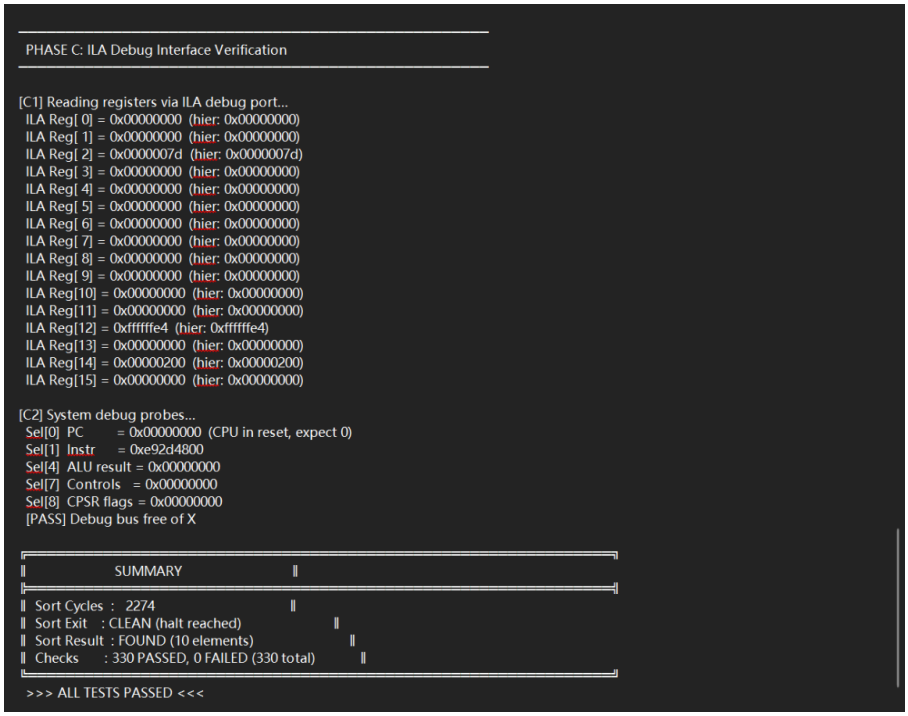


Figure 5: Representative pipeline waveform / ILA probe capture

11. Part 2: Quad Hardware Threading (Implementation Details)

Lab 6 Part 2 requires a hardware-augmented processor with four threads supported by multiple PCs and a multithreaded register file, enabling low-overhead context switching. We successfully implemented a fine-grained hardware multithreading architecture (Barrel Processing) in our `cpu_mt.v` module that achieves true zero-overhead context switching.

11.1 HARDWARE DUPLICATION AND CONTEXT ISOLATION

To support the concurrent execution of four independent network-processing threads, we physically replicated the core states in the datapath:

- **Multiple Program Counters:** We instantiated four independent PCs (pc_thread[0:3]). During the IF stage, the fetch address is dynamically selected based on the current thread ID (tid_if).
- **Multithreaded Register File:** Using Verilog generate blocks, we instantiated four distinct 16-word register files (THREAD_RF[0:3]). Write operations in the WB and MEM stages are properly demultiplexed to the correct target thread using the pipeline's thread ID, ensuring strict physical isolation of contexts.
- **Independent CPSR Flags:** We implemented four isolated sets of condition flags (N, Z, C, V) to ensure that ALU operations or branching in one thread do not interfere with the conditional execution of others.

11.2 ZERO-OVERHEAD CONTEXT SWITCHING SCHEDULER

Unlike software context switching, our processor incurs zero cycle overhead. We implemented a continuous Round-Robin Scheduler at the IF stage. Unless the shared memory bus is stalled by the Block Data Transfer Unit (BDTU), the thread ID increments every clock cycle (tid_if <= tid_if + 1), wrapping from 0 to 3. This thread ID seamlessly propagates down the pipeline registers (tid_id, tid_ex, tid_mem, tid_wb), meaning the five pipeline stages process instructions from up to four different threads simultaneously.

11.3 HAZARD ELIMINATION VIA INTERLEAVING (ARCHITECTURAL OPTIMIZATION)

The most significant architectural advantage of our quad-threaded design is the natural elimination of data and control hazards. In our single-core implementation (cpu.v), a complex Forwarding Unit (fu.v) and Hazard Detection Unit (hdu.v) were required to handle data dependencies.

However, in our cpu_mt.v design, instructions from the same thread are separated by four clock cycles due to the 4-thread interleaving. By the time Thread 0 fetches its N+1 instruction, its N instruction has already reached the WB stage. This physical time gap completely resolves Read-After-Write (RAW) data hazards and branch delays. Consequently, we were able to entirely eliminate the Forwarding Unit (fu.v) from the multithreaded datapath and significantly simplify the HDU, relying purely on the multithreading interleaving to maintain pipeline efficiency without stalling.

Important implementation note (shared-resource limitation):
All four threads share the same memory interface and the same multi-cycle engine; therefore, when BDTU is active (bdtu_busy=1), the core asserts a global stall (stall_all) that freezes thread scheduling until the multi-cycle sequence completes. This does not affect correctness, but it bounds the degree to which multithreading can hide latency for multi-cycle memory sequences.

11.4 MULTITHREADED SOFTWARE EXAMPLES (NETWORK_PROC0–3)

To demonstrate realistic “network processing” style workloads, we prepared four independent C programs and compiled each into ARM machine code:

- Thread 0: Packet header/NAT + checksum (network_proc0.c)

Validates packet type, swaps src/dst IP fields, and computes a payload checksum.

- Thread 1: XOR encryption (network_proc1.c)

If status is “plain”, XOR-encrypts four payload words with a constant key and updates status to “secure”.

- Thread 2: Router TTL processing (network_proc2.c)

Drops packets with TTL ≤ 1; otherwise decrements TTL and sets action to forward.

- Thread 3: Firewall port filter (network_proc3.c)

Rejects traffic with a blocked destination port; otherwise accepts.

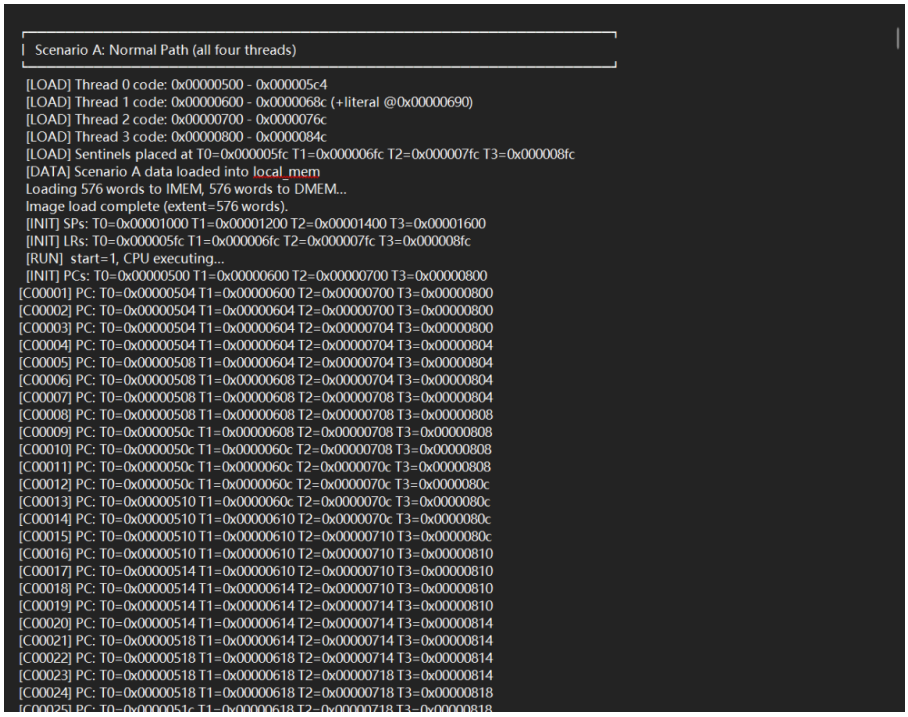
In our multithreaded testbench trace, we load these programs at distinct IMEM base addresses (0x0000, 0x1000, 0x2000, 0x3000), initialize per-thread stacks and data regions, and then observe round-robin execution across threads with the expected memory updates.

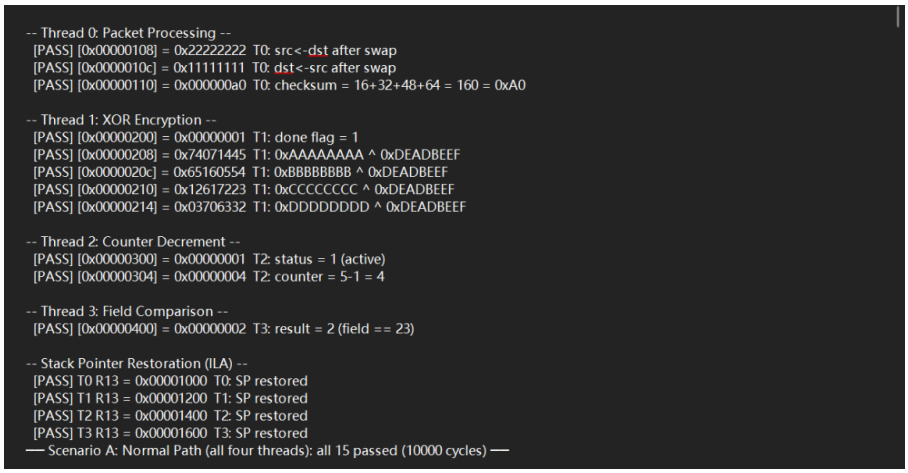
12. Conclusion

We implemented a 5-stage pipelined ARM-style CPU core capable of executing a substantial ARMv4-like instruction subset required by compiler-generated sorting programs, including conditional execution, barrel shifting, load/store addressing modes, branch/link/return, multiply, and multi-cycle block transfer support. We integrated the core into a NetFPGA-friendly SoC with an MMIO programming interface and a lightweight debug/probe mechanism.

Furthermore, we successfully upgraded the core to a Quad HW-Threaded architecture, demonstrating (via our multithreaded verification flow) that multiple network packet-processing threads can execute concurrently with zero context-switching overhead, effectively masking memory latencies for typical single-cycle operations and elegantly eliminating the need for

complex data forwarding logic.





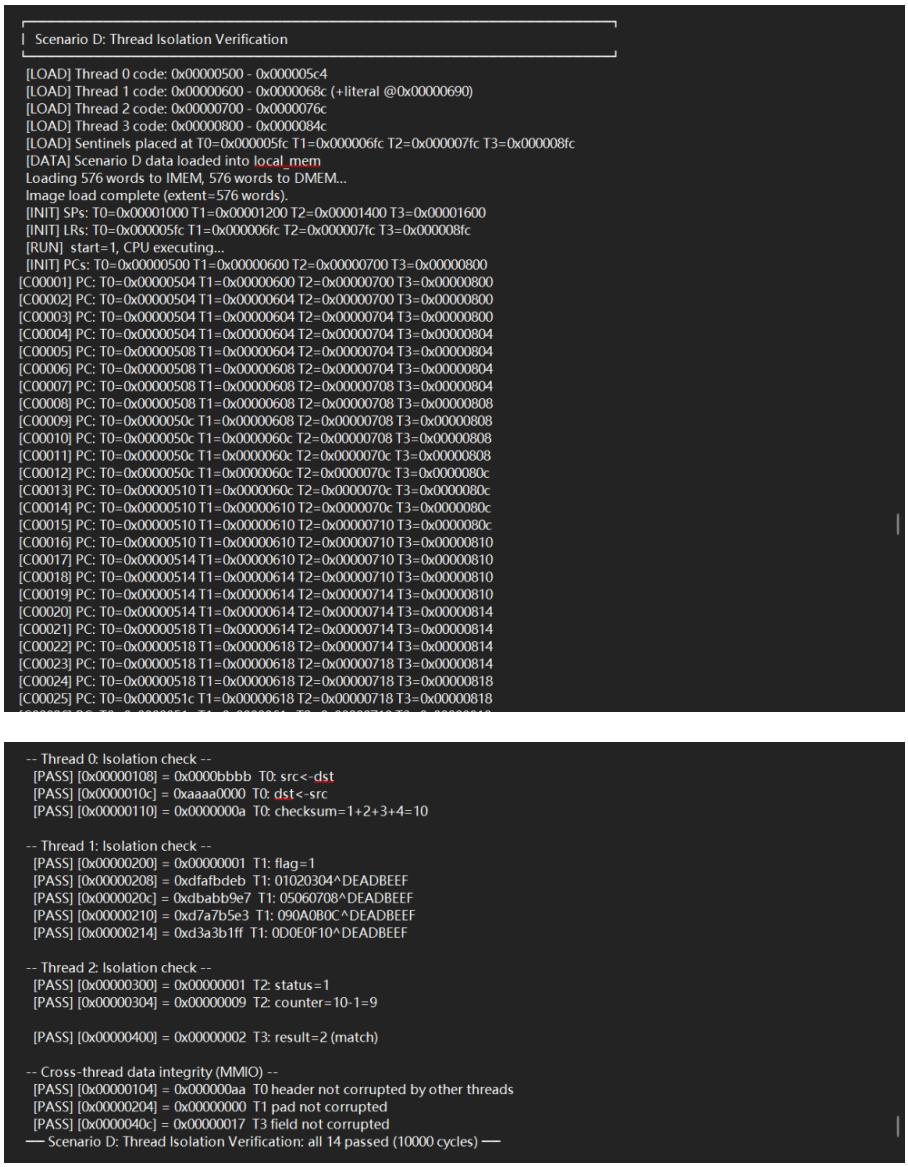


Figure 6: Multithreaded execution trace / ILA snapshot

13. GitHub Repository and Team Contributions

GitHub Repository: EE533-Labs/Lab6/src at main · jeremycai99/EE533-Labs
Contribution Summary:

- Jeremy Cai: Implemented the single-thread ARM-like 5-stage CPU (cpu.v) including decode/cond-exec, hazard/forwarding, and BDTU multi-cycle support; Verified bubble-sort execution and core functional tests.
- Prathamesh Hirekodi: Built the SoC/MMIO transaction path (IMEM/CTRL/DMEM decode, req/resp handshake, run-time busy gating); Wrote assembler.sh and supported IMEM/DMEM loading + system-level debug logs.
- Yanzhe Li: Implemented the quad HW-threaded core (cpu_mt.v) with per-thread PC/RF/flags and round-robin scheduling; Prepared multithread workloads (network_proc0–3) and validated thread isolation/debug traces.