EE533: Network Processor Design & Programming

Lab #9: Design and Integration of Hardware for Multiple Threads

Instructor: Prof. Young Cho, PhD

Team Number: **#3**

Project Partners:

Member #1: **Sarthak Jain**

Member #2: **Archit Sethi**

Member #3: **Justin Santos**

Designed, Created, and Submitted by Team #3

University of Southern California

Los Angeles, CA 90007

Team #4 GitHub Repository Link: *Team#3 Link*

1. How does out design support 4 threads in parallel?
   Ans: Our design integrates a multi-threaded hardware architecture that enable execution of 4 independent threads with minimal overhead. This is achieved using the following hardware modifications:
   a. **Multi-threaded Register File:**
      i. The RF is designed using block memory (BRAM) instead of separate flip-flop-based registers.
      ii. The upper 2-bits of the memory address are used to index the thread.
      iii. The remaining bits in the address specify register selection

      iv. Why this works efficiently because of the following reasons:
         1. The memory-based RF reduces the hardware throughput.
         2. Efficient indexing allows simultaneous access to multiple registers.
   b. Multi-threaded PC (Program Counter):
      Each thread maintains a separate PC enabling context switching without needing explicit saving/restoring.

      Each Core Has a Separate Program Counter (PC)

      Each core will start executing from a different instruction memory address.
      Instruction Memory Supports Different Programs

      We can either:
      Have one shared instruction memory where each core starts at different offsets.
      Have two separate instruction memories (one per core) for complete isolation.
      Shared Data Memory with a Bus Arbiter

      Both cores will access shared data memory for load/store operations.
      A bus arbiter will handle memory conflicts.

      Before diving into the Verilog, let's outline the minimal required modules:

      Core Module (core.v):

      Contains the 5-stage pipeline
      Has a PC register, instruction fetch unit, and pipeline registers
      Requests memory access from the shared bus
      Instruction Memory (imem.v):

Read-only memory storing instructions for both cores
Simple implementation as a register array or ROM
Data Memory (dmem.v):

Shared memory for storing data
Needs a bus arbiter to resolve conflicts between cores
Bus Arbiter (bus_arbiter.v):

Controls access to the shared data memory
Implements a round-robin scheduling policy
Top Module (multicore_cpu.v):

Instantiates two cores, shared memory, and the bus arbiter
Handles system-wide clock and reset signals

For design choice:
4 separate PC registers mapped to thread IDs and a multiplexer (MUX) selects the correct PC based on the active thread.

c. Scheduling Mechanism:
    i. The processor scheduler selects one of the 4 threads each cycle.
    ii. Thread switching takes place without OS involvement which reduces the overhead.
d. Instruction Execution Model
   During each cycle the processor selects the next thread round-robin which ensures fair execution.

2. How did we modify the register file to support multiple threads?
   The RF is built using block memory (BRAM) instead of traditional flip-flop-based registers allowing efficient access to thread-specific registers.

    a. Major modifications:
        i. The upper 2-bits of the memory address resemble the thread ID.
        ii. The remaining bits are used to specify the register address.
        iii. Each thread accesses its dedicated register set which avoids data corruption.

    b. Handling Read/Write Ports
        i. Since BRAM allows only 2 thread read/write ports, implement a double-clocking mechanism:
            1. 1st clock cycle: Read operation

2. 2nd clock cycle: Write Operation

3. How does the hardware ensure minimal overhead in context switching?
   Ans: Each thread has a dedicated PC and register file segment. Instead of saving/restoring during context switching, we implemented a thread indexing. No extra cycles were wasted on saving/restoring context.

   The result is that the system switches thread instantly without pipeline flushing.
   a. Discuss the method used for saving/restoring thread state.
   b. If you implement a zero-overhead thread switching mechanism, explain how it was achieved
4. How were the 4 independent network packet processing programs designed in C?
   Each program is designed to process network packets in parallel.
   Consider programs:

   Thread 1: Extracts packet headers and classifies packets.
   Thread 2: Computes checksums for packet integrity verification
   Thread 3: Filters malicious traffic based on IP blacklists.
   Thread 4: Forwards packets to different queues based on priority.

5. How does your processor execute the programs in parallel?
   The processor assigns each thread a dedicated execution cycle.
   It follows a round-robin scheduling algorithm.
   Each thread fetches and executes instructions independently.

   Parallel Execution Mechanism:

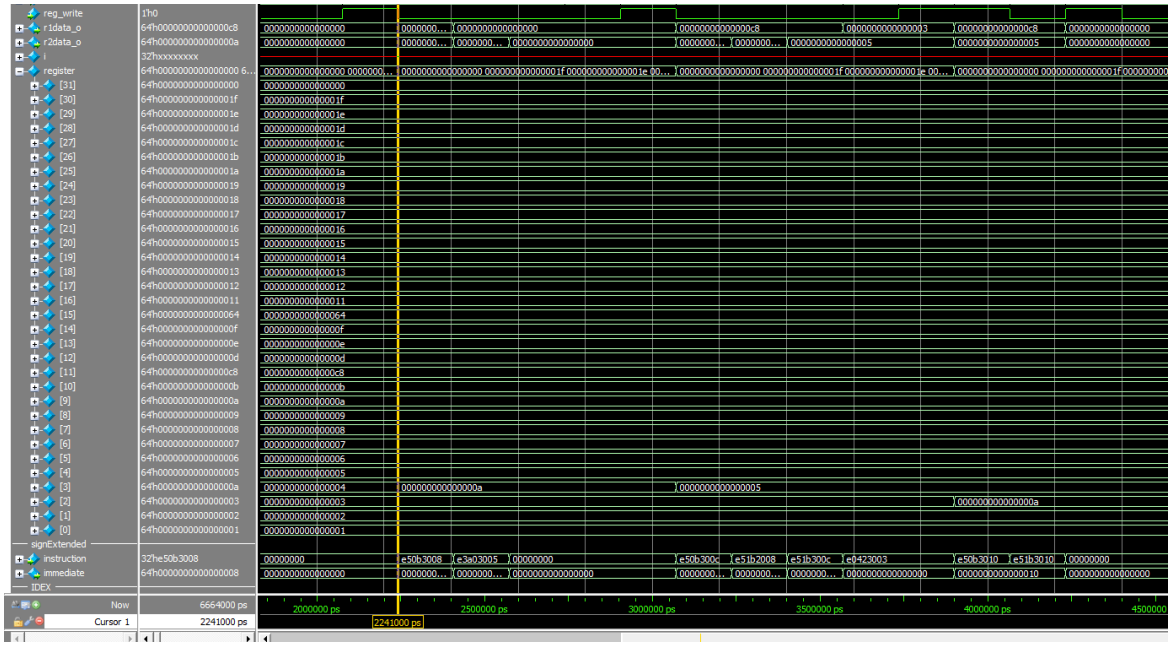
Cycle	**Thread Executing**
Cycle 1	Thread 1
Cycle 2	Thread 2
Cycle 3	Thread 3
Cycle 4	Thread 4
Cycle 5	Thread 1 (Next)

6. How did you prove that the software works correctly with the hardware?
   Simulation logs: Verified that each thread executed in the expected order.

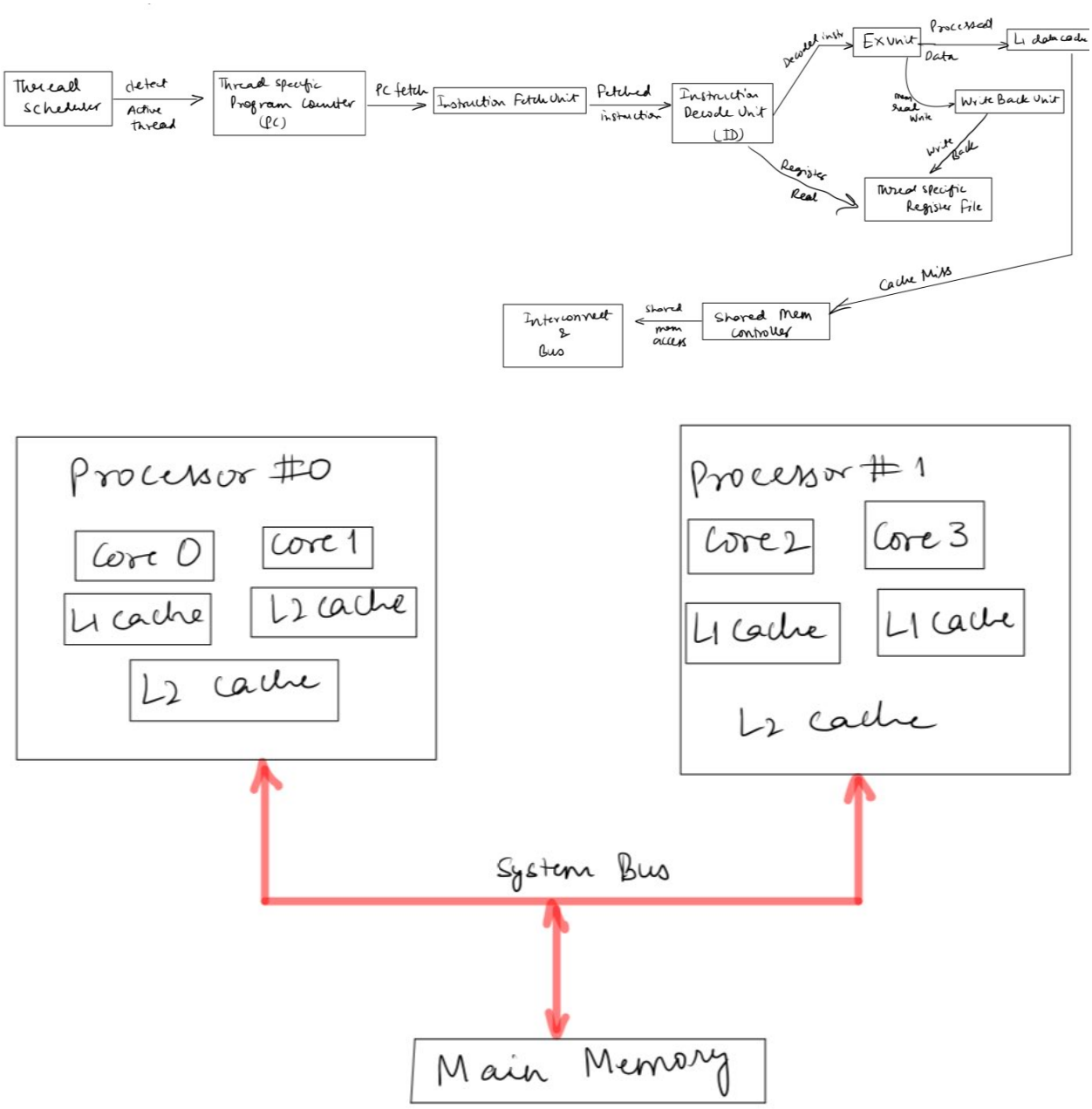
```
[team-3:fpga ARMv7] ./lab7_pipeline read_dmem 192
Found net device: nf2c0
Found net device: nf2c0
Data at address 192 is: 0x000000000000000a
[team-3:fpga ARMv7] ./lab7_pipeline read_dmem 188
Found net device: nf2c0
Found net device: nf2c0
Data at address 188 is: 0x0000000000000005
[team-3:fpga ARMv7] ./lab7_pipeline read_dmem 184
Found net device: nf2c0
Found net device: nf2c0
Data at address 184 is: 0x000000000000000f
```

   Waveform analysis: Checked if RF accesses corresponded to the correct threads.
   Output verification: Each thread's output matches expected results.

   a. Mention the test cases and verification methods used.
   b. Show evidence of successful execution (e.g. waveform)
      Register File getting populated properly.



7. Provide a high-level diagram of the datapath. Draw and label RF, Thread PC counters, Memory interface, ALU, Thread selection logic.



We used a single bus for memory access, and independent execution of instructions. We also used a basic round-robin scheduler to manage execution.

High-Level Design Steps:
Modify Your Existing Pipeline to Support Multiple Cores

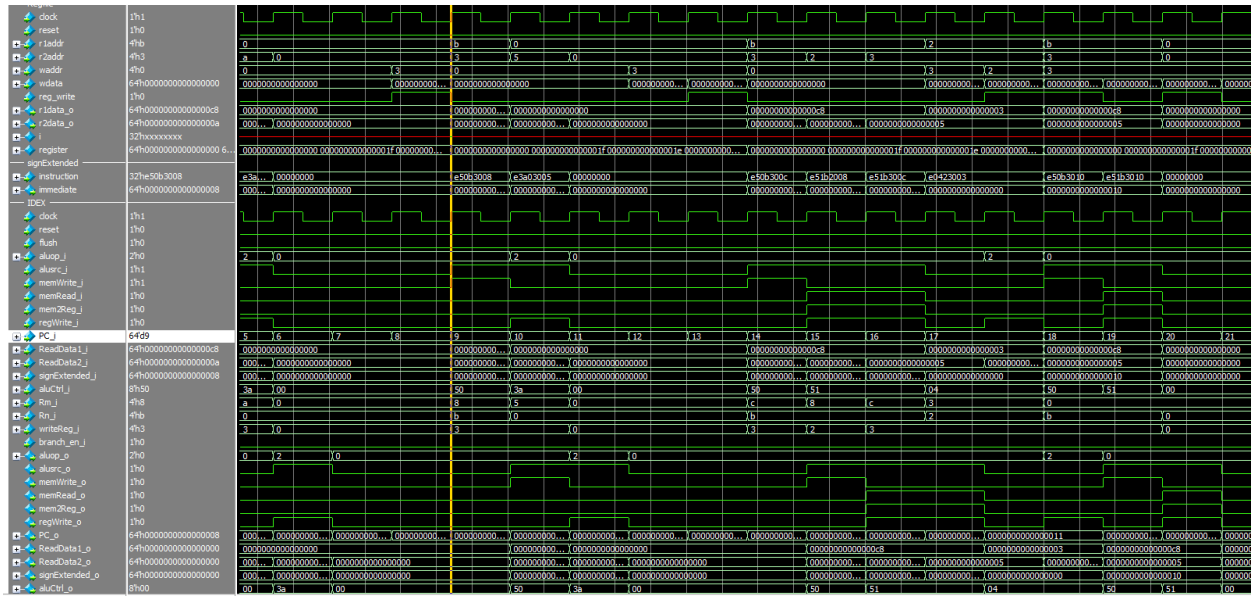
Duplicate your existing 5-stage pipeline into two instances (Core 0 and Core 1).
Each core should have its own program counter (PC) and pipeline registers.
Implement Shared Instruction Memory and Data Memory

Instruction memory (IMEM) should be read-only and shared between cores.
Data memory (DMEM) should be shared, requiring a bus arbitration mechanism to handle memory conflicts.
Design a Simple Bus Arbiter

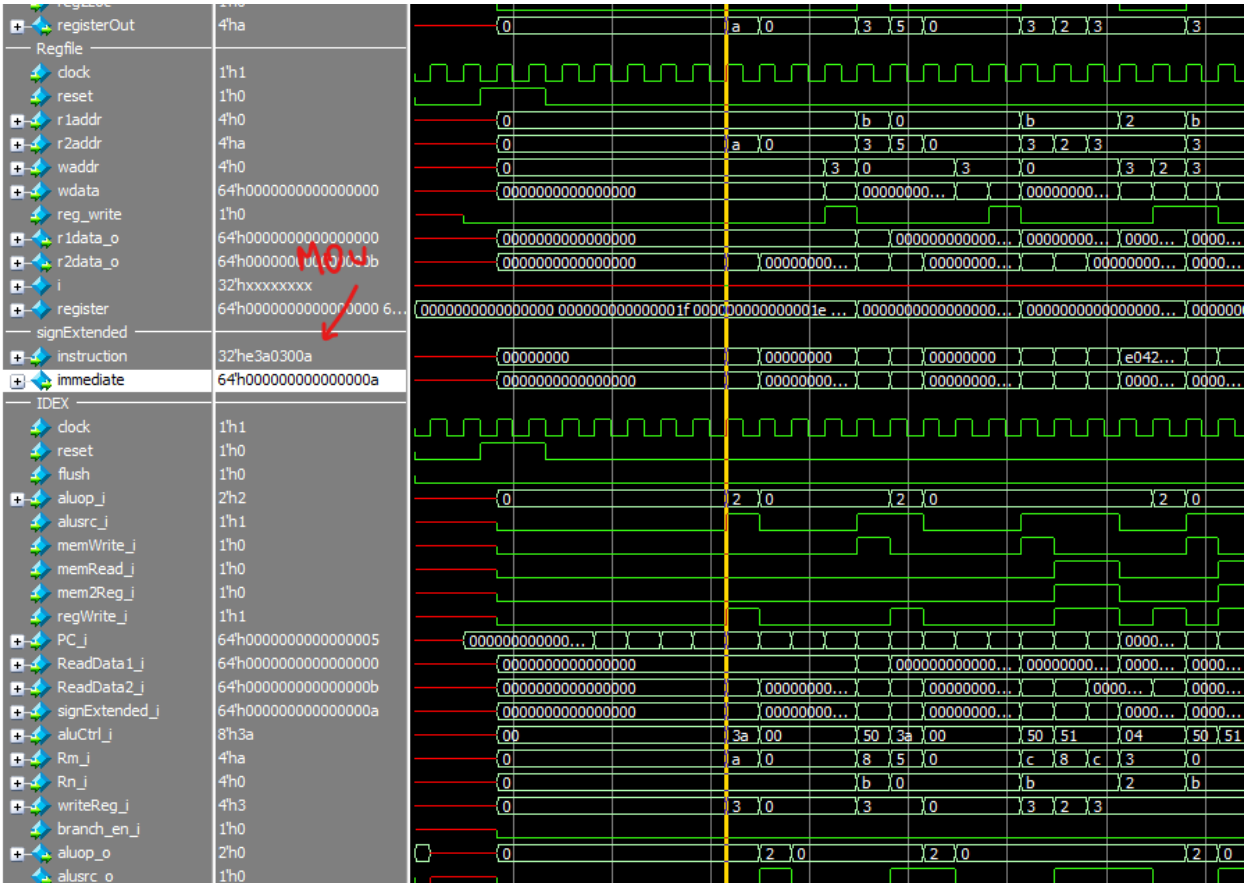
Since both cores access shared memory, a bus arbiter should resolve conflicts.
Use a round-robin scheduler to alternate memory access between cores.
Ensure Independent Execution

Each core should fetch its own instructions independently.
The cores should not interfere unless explicitly designed to do so.

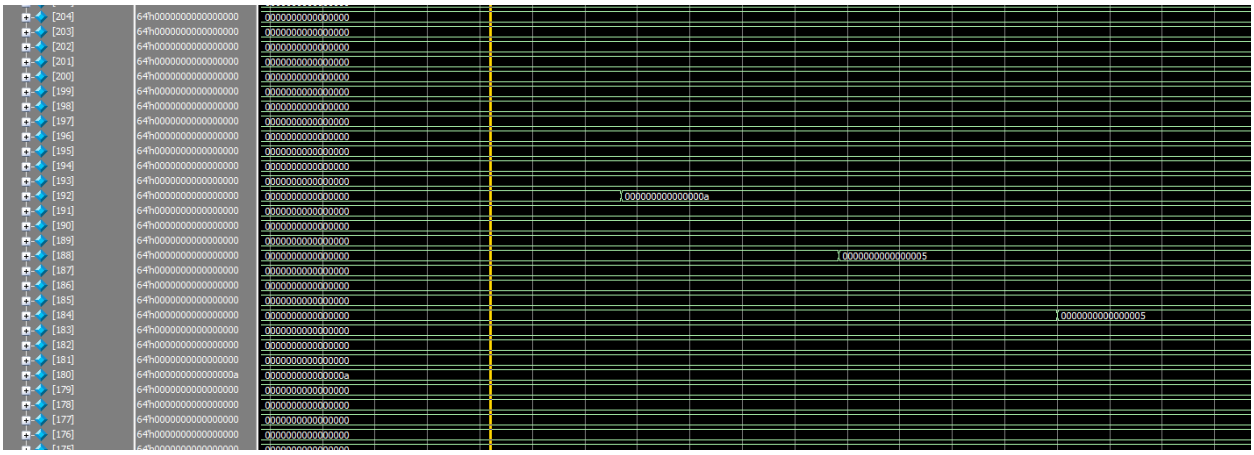
8. Provide screenshots of hardware schematics

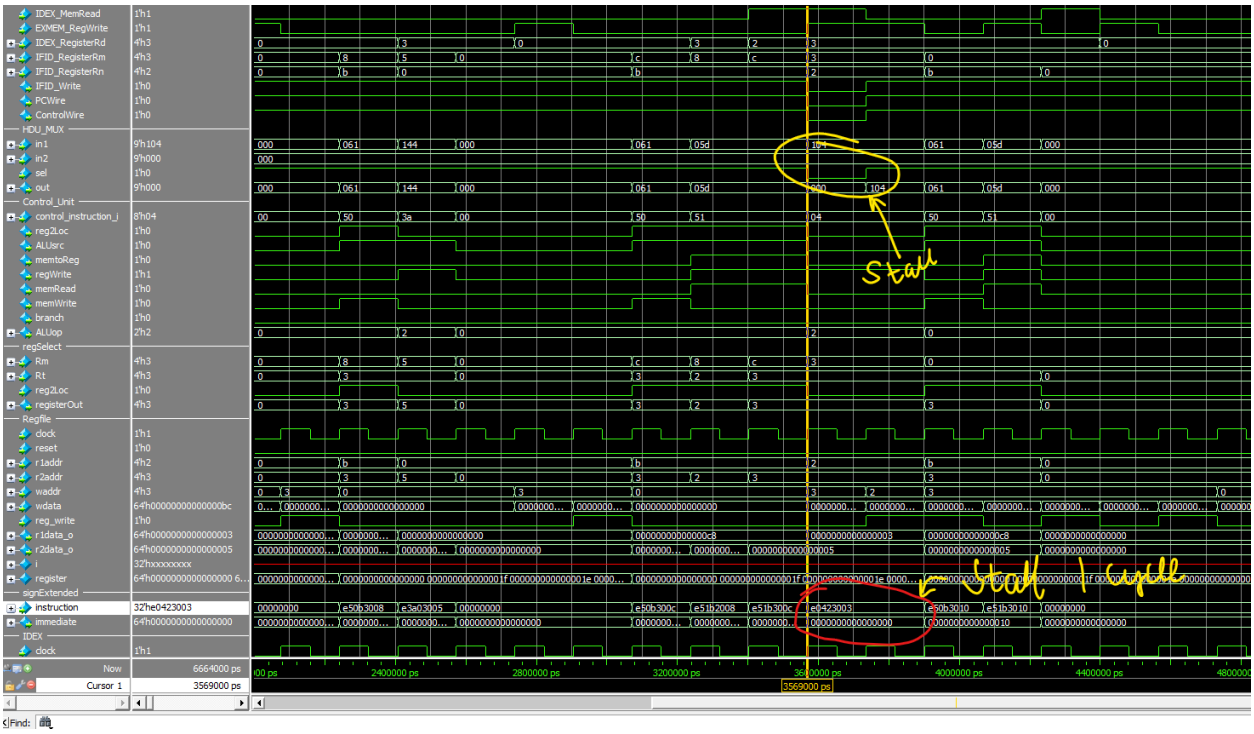


MOV Instruction:



Data memory (DM) getting filled based on instructions





We ran a simple add on our processor, supporting armv7 ISA. This code performs 5+10=15. Our processor has forwarding and hazard detection for all instructions except MOV, hence we placed NOPs after every MOV.

1. e3a0300a // MOV R3, #10 = Set R3 to 10
2. 00000000
3. 00000000
4. 00000000
5. e50b3008 // STR R3, [R11, #-8] = Store 10 into DMEM
6. e3a03005 // MOV R3, #5 = Set R3 to 5
7. 00000000
8. 00000000
9. 00000000
10. e50b300c // STR R3, [R11, #-12] = Store 5 into DMEM
11. e51b2008 // LDR R2, [R11, #-8] = Load 10 into R2
12. e51b300c // LDR R3, [R11, #-12] = Load 5 into R3
13. e0423003 // SUB R3, R2, R3 = Add 10+5

14. e50b3010 // STR R3, [R11, #-16] = Store 10+5 to DMEM

15. e51b3010 // LDR R3, [R11, #-16] = Load 10+5 into R3