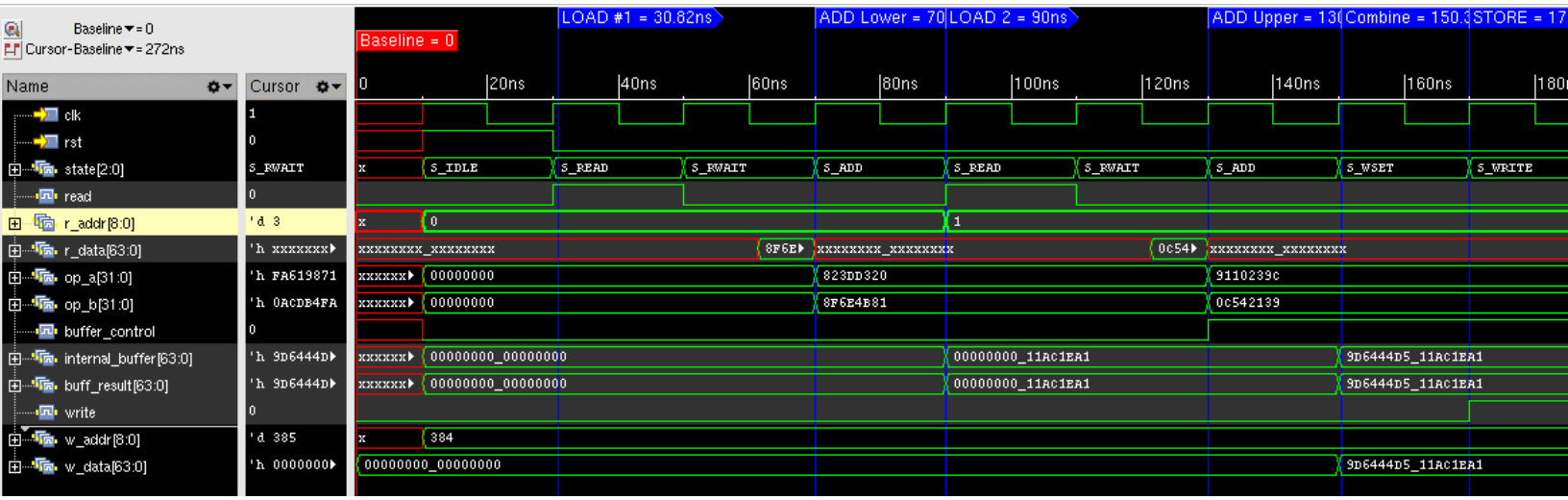
```
Time:             2061000
Simulation complete via $finish(1) at time 20610 NS + 0
./sym_links/tb_calculator.sv:52        $finish;
xcelium> exit

coverage setup:
  workdir  :  ./cov_work
  dutinst  :  tb_calculator(tb_calculator)
  scope    :  scope
  testname :  test

coverage files:
  model(design data) :  ./cov_work/scope/icc_7019b285_00000000.ucm
  data               :  ./cov_work/scope/test/icc_7019b285_00000000.ucd
TOOL:   xrun(64)       25.03-s003: Exiting on Sep 11, 2025 at 13:58:29 EDT  (total: 00:00:00)
make[1]: Leaving directory '/nethome/skujuluva3/Digital-Design-Onboarding/sim/behav'
python3.12 ../../scripts/check_onboarding.py /nethome/skujuluva3/Digital-Design-Onboarding/sim/behav/WORKSPACE/memory_post_state_lower.txt /nethome/skujuluva3/Digital-Design-Onboarding/sim/behav/WORKSPACE/sim_memory_post_state_lower.txt
PASSED: RTL simulation output matches expected results.
python3.12 ../../scripts/check_onboarding.py /nethome/skujuluva3/Digital-Design-Onboarding/sim/behav/WORKSPACE/memory_post_state_upper.txt /nethome/skujuluva3/Digital-Design-Onboarding/sim/behav/WORKSPACE/sim_memory_post_state_upper.txt
PASSED: RTL simulation output matches expected results.
```



My controller is a small FSM that walks memory in order. S_IDLE initializes the read/write pointers. S_READ asserts read to fetch a 64-bit input word. S_RWAIT basically burns one cycle for SRAM latency so r_data_q is stable. S_ADD toggles a “half” so that the first pass (half=0) we process is the lower half and the second pass (half=1) we process the upper half. After the upper pass we move to the write path and we have all 64 bits. S_WSET drives w_addr and w_data (= buff_result) with write=0 so address/data settle, and S_WRITE pulses write=1 on the next edge to commit the store and then bumps the pointers (+1). We repeat until the end addresses, then finish in S_END. This sequencing matches the SRAM’s read latency and write setup, so we never sample/commit on unstable signals.

The “calculator” adds by splitting each 64-bit read word into two DATA_W-wide operands: op_a = r_data_q[DATA_W-1:0] and op_b = r_data_q[2*DATA_W-1:DATA_W]. In the first visit to S_ADD we set buffer_control=0 to capture the lower-half sum into a tiny result buffer. Then we read again and on the next S_ADD with buffer_control=1, we capture the upper-half sum (same logic as half). The result buffer combines two DATA_W results (the two 32-bit sums) into one buff_result that is MEM_WORD_SIZE 64 bit wide. After both halves are produced, S_WSET/ S_WRITE write the combined word to the output address. Because every write is preceded by exactly two 32 bit adds on the corresponding input word—and pointers only advance within the specified ranges—the design is functionally correct and covers the whole window.