0512.4490 Advanced Computer Architecture Lab 2021

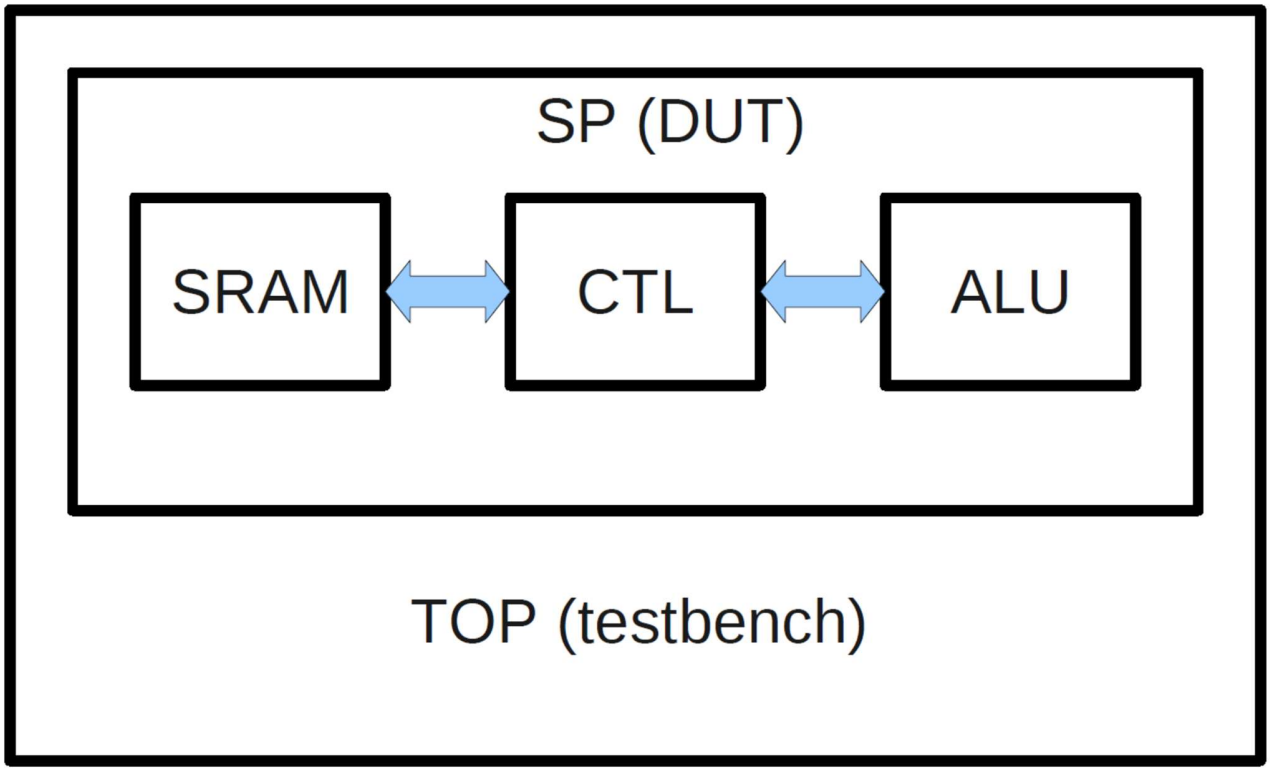
Lab #4: Processor Verilog Implementation

In the previous labs, we designed a high-level instruction set architecture simulator, and a low-level micro architecture cycle accurate simulator for the SP processor. In this lab, we'll complete the design cycle by implementing the processor micro architecture in the Verilog hardware description language and perform verification against the low-level simulator which we designed in lab #2.

The high-level ISS simulator generated an instruction trace file. The low-level simulator generated two trace files: an instruction trace, which was binary compared to the ISS trace to verify the match between high and low level simulators, as well as a cycle trace.

In the lab today, the Verilog implementation will output an identical cycle trace, which we will binary compare against the cycle trace from the low-level simulator to verify the hardware implementation matches the simulation.

The implementation contains the following hierarchical Verilog modules:



SRAM memory (defined in sram.v)

The SRAM contains a Verilog behavioral model for a synchronous SRAM, 65536 lines x 32 bit each.

Inputs:

clk: system clock, the SRAM is sensitive to the positive edge of the clock.
addr[15:0]: 16 bit address.
di[31:0]: 32 bits data input.
en: 1 bit enable signal.
we: 1 bit write enable: 1 for write, 0 for read.

Outputs: do[31:0]: data out.

ALU (defined in alu.v)

The ALU contains combinatorial logic implementing ADD, SUB, LSF, RSF, AND, OR, XOR, LHI, JLT, JLE, JEQ and JNE. It accepts as inputs:

opcode[4:0]: specifying the operation type.
alu0[31:0]: 32-bit first operand.
alu1[31:0]: 32-bit second operand.

And provides as output alout[31:0].

CTL (control logic, defined in ctl.v)

The ctl module contains the micro-architecture registers described in lab #4, and the implementation of the control state machine.

Inputs:

reset: reset signal, active high.
clk: system clock
start: '1' to activate the processor.
sram_DO[31:0]: memory outputs from SRAM module.
aluout_wire[31:0]: from ALU.

Outputs:

sram_ADDR[15:0]: memory address.
sram_DI[31:0]: memory data in.
sram_EN: memory enable
sram_WE: memory write enable
alu0[31:0], alu1[31:0]: ALU operands.
opcode[4:0] opcode for ALU.

SP (simple processor module, defined in sp.v): SP is a parent module containing the SRAM, ALU and CTL modules.
TOP (top level testbench, defined in top.v): TOP is the top level module containing the testbench, and instantiating SP as the DUT (design under test).

Question 1: SRAM verification

Fill the missing code in the sram.v module, which models the SRAM, and in the sram_tb.v module, which is the testbench for the SRAM. Verify the SRAM, and submit sram.v, sram_tb.v, and a waveform screenshot showing a successful memory write transaction followed by a read transaction of the same data.

Question 2: processor implementation (no DMA)

Fill the missing ALU, SP and CTL code in alu.v, sp.v and ctl.v. Submit your modified files.

Question 3: verification #1: example.bin

Run a simulation of the provided example.bin. Compare the generated verilog_trace.txt to the provided example and make sure it matches.

Question 4: verification #2: multiplying two signed numbers

Run a simulation of the same program which multiplies two signed numbers from labs #1 and #2 (rename your bin file to example.bin). Compare verilog_trace.txt to the cycle trace of the low-level simulator from lab #2. Write in your report the results of this comparison. Submit mult_verilog_trace.txt, and the cycle trace from lab#2 and two waveform screenshots: one showing the numbers being read into registers from memory, and one showing the correct result being written back to memory.

Question 5: verification #3: Fibonacci

Run a simulation of the same program which generates the Fibonacci sequence from labs #1 and #2 (rename your bin file to example.bin). Compare verilog_trace.txt to the cycle trace of the low-level simulator from lab #2. Write in your report the results of this comparison. Submit fibo_verilog_trace.txt, and the cycle trace from lab#2 and two waveform screenshots showing the last two writes to memory (the writes of the last two values of the Fibonacci sequence).

Question 6: DMA implementation

Add the DMA state machine you designed in lab #2 and submit the modified files.

Question 7: DMA verification

Verify your DMA design by running the DMA test program from lab #2 and comparing the trace files. Submit:

1. In the report: a waveform screenshot containing memory access of both the assembly program, and the DMA machine, demonstrating structural hazard detection and resolution.
2. Files:
   - `dma_verilog_trace.txt` – the trace of the execution
   - `dma_sram_out.txt` – the memory output image