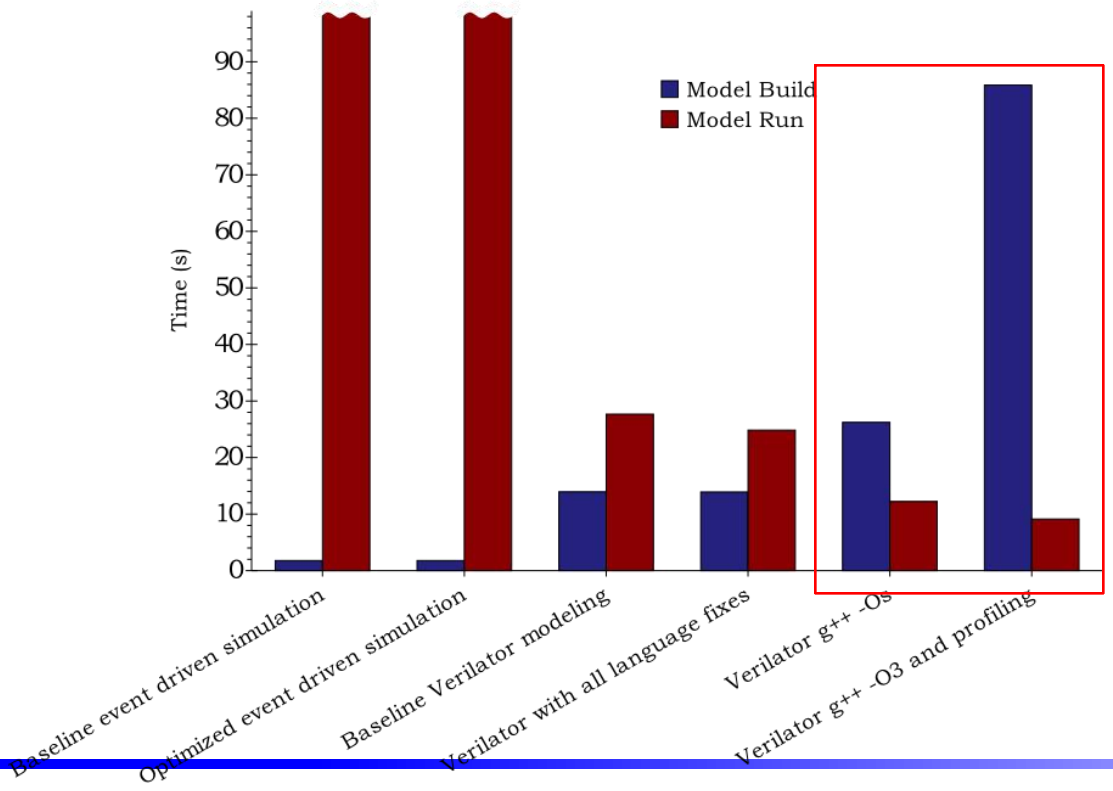
编辑母版文本样式

FULL CYCLE SIMULATION USING VERILATOR

Introduction to Verilator

- Verilator is the state-of-the-art RTL simulator
  - support a lot of SystemVerilog feature
  - compile Verilog / SystemVerilog to C++



Verilator Performance

This LAB

- This lab is divided into 4 parts
  - use verilator to lint code
  - write a simple C++ testbench
  - use verilator to run coverage test
  - build large design and tune compilation options

Install depedency

- `apt install npm lcov cloc`
- `npm install light-server`

Part. 1. Lint

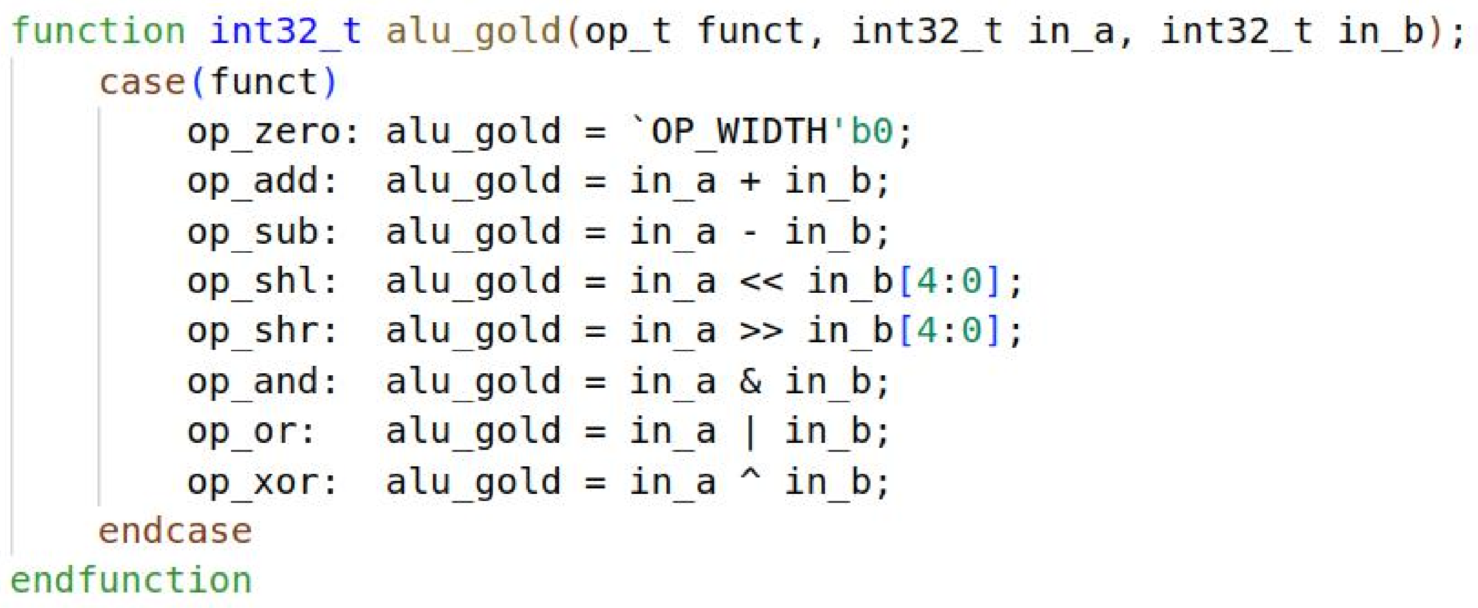
- In `01-simple`, the ‘alu.sv’ has some typos

```
verilator --lint-only alu.sv
```

  – check the output and fix the typos

Part. 2. A Simple C++ Testbench

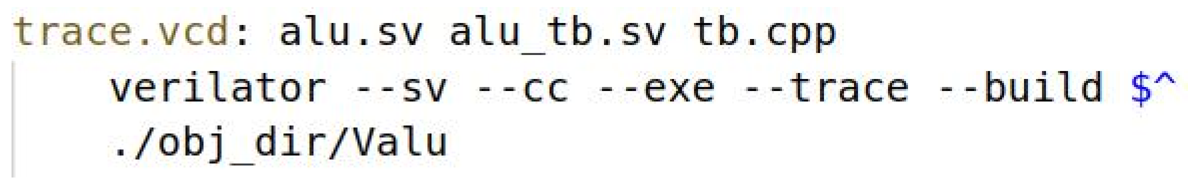
- The testbench is a simple alu
  - the detailed implementation is in alu.sv
  - the functional equivalent module is in alu_tb.sv



```
function int32_t alu_gold(op_t funct, int32_t in_a, int32_t in_b);
    case(funct)
        op_zero: alu_gold = `OP_WIDTH'b0;
        op_add:  alu_gold = in_a + in_b;
        op_sub:  alu_gold = in_a - in_b;
        op_shl:  alu_gold = in_a << in_b[4:0];
        op_shr:  alu_gold = in_a >> in_b[4:0];
        op_and:  alu_gold = in_a & in_b;
        op_or:   alu_gold = in_a | in_b;
        op_xor:  alu_gold = in_a ^ in_b;
    endcase
endfunction
```

Part. 2. A Simple C++ Testbench

- Add verilator compile command to Makefile

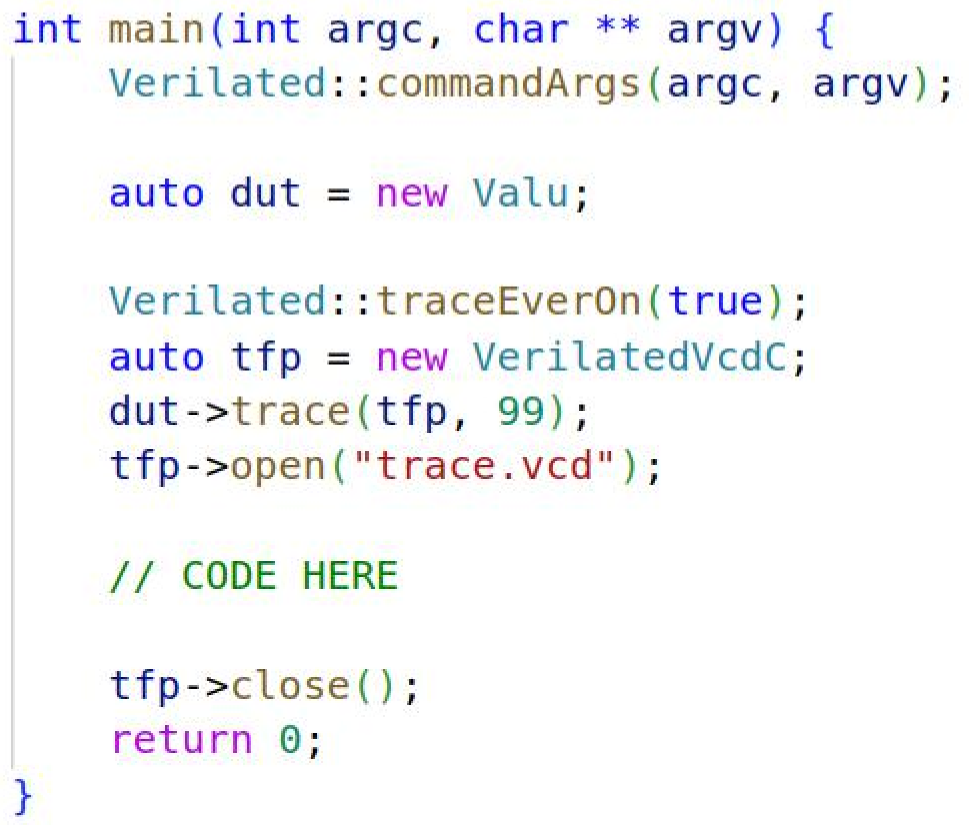


```
trace.vcd: alu.sv alu_tb.sv tb.cpp
	verilator --sv --cc --exe --trace --build $^
	./obj_dir/Valu
```

  - `--sv`: this is a sv module
  - `--cc`: compile to cpp
  - `--exe`: build executable, not library
  - `--build`: build after emit
  - `--trace`: enable signal trace
  - `$^`: means the `alu.sv alu_tb.sv tb.cpp`
    - the order matters!

Part. 2. Simple C++ Testbench

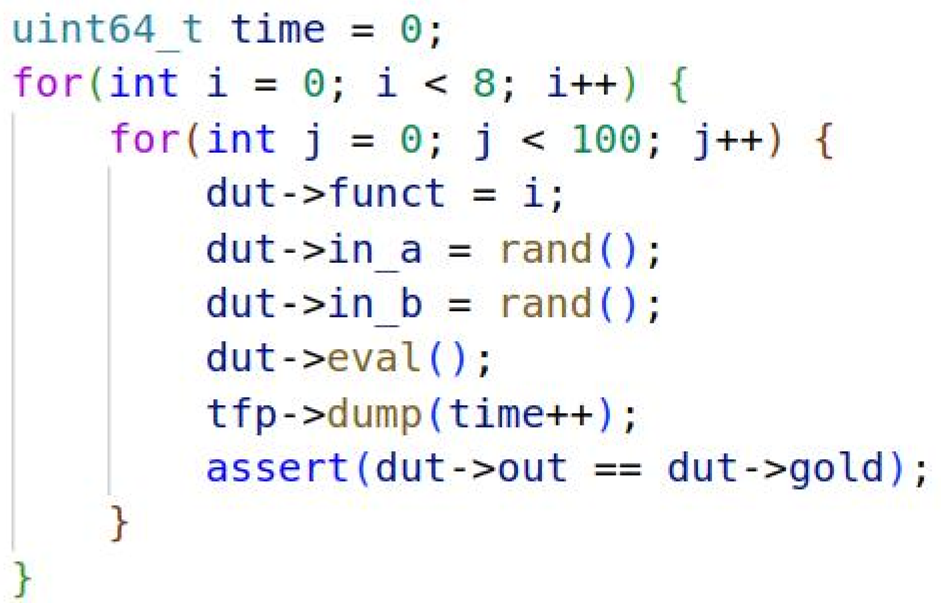
- First tell verilator the command args
- Instanciate the target module
- Open trace file
- Run test
- Close trace file



```cpp
int main(int argc, char ** argv) {
    Verilated::commandArgs(argc, argv);

    auto dut = new Valu;

    Verilated::traceEverOn(true);
    auto tfp = new VerilatedVcdC;
    dut->trace(tfp, 99);
    tfp->open("trace.vcd");

    // CODE HERE

    tfp->close();
    return 0;
}
```

Part. 2. A Simple C++ Testbench

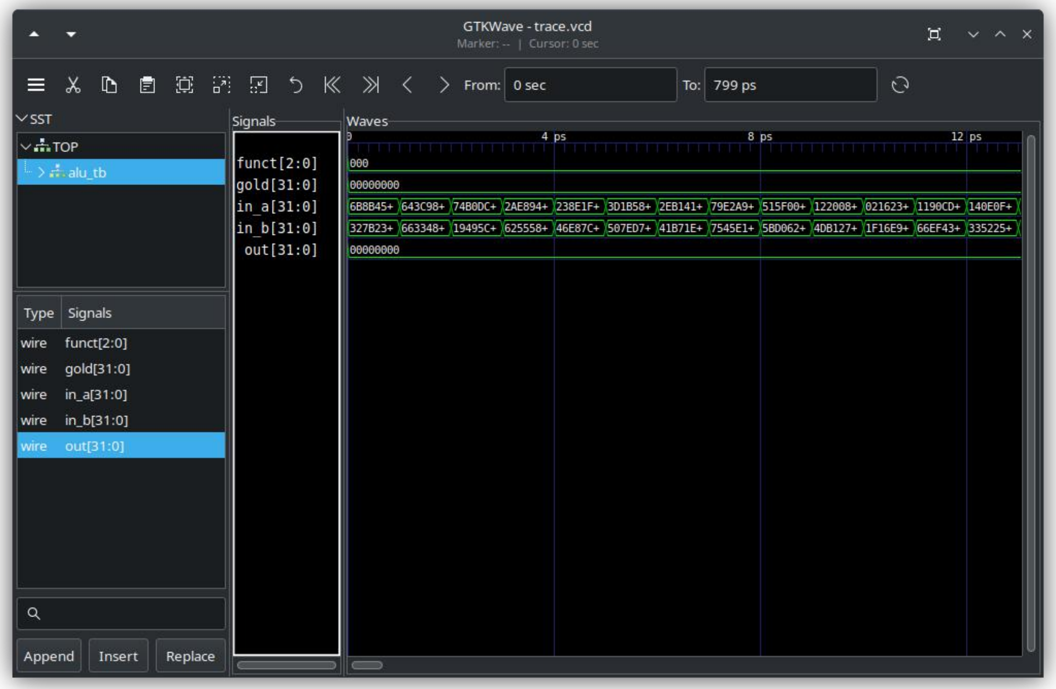
- `make`, to ensure no compilation error
- Then, we enumerate all `funct`, and generate test input
- unlike SystemC, verilator has no timing information
  - the time cycle is user managed `tfp->dump(time++)`



```
uint64_t time = 0;
for(int i = 0; i < 8; i++) {
    for(int j = 0; j < 100; j++) {
        dut->funct = i;
        dut->in_a = rand();
        dut->in_b = rand();
        dut->eval();
        tfp->dump(time++);
        assert(dut->out == dut->gold);
    }
}
```

Part. 2. A Simple C++ Testbench

- `make` to compile and run
- `make show` to show the waveform

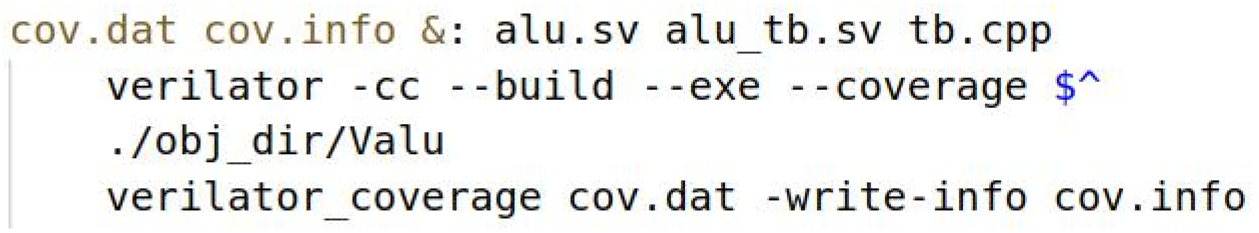


```
Type  Signals
wire  funct[2:0]
wire  gold[31:0]
wire  in_a[31:0]
wire  in_b[31:0]
wire  out[31:0]
```

```
            0                           4 ps                                8 ps                              12 ps
funct[2:0]  000
gold[31:0]  00000000
in_a[31:0]  6B8B45+ 643C98+ 74B0DC+ 2AE894+ 238E1F+ 3D1B58+ 2EB141+ 79E2A9+ 515F00+ 122008+ 021623+ 1190CD+ 140E0F+
in_b[31:0]  327B23+ 663348+ 19495C+ 625558+ 46E87C+ 507ED7+ 41B71E+ 7545E1+ 5BD062+ 4DB127+ 1F16E9+ 66EF43+ 335225+
out[31:0]   00000000
```

Part. 3. Coverage Test

- In `02-coverage`, the design is the same as 01
- Modify the Makefile

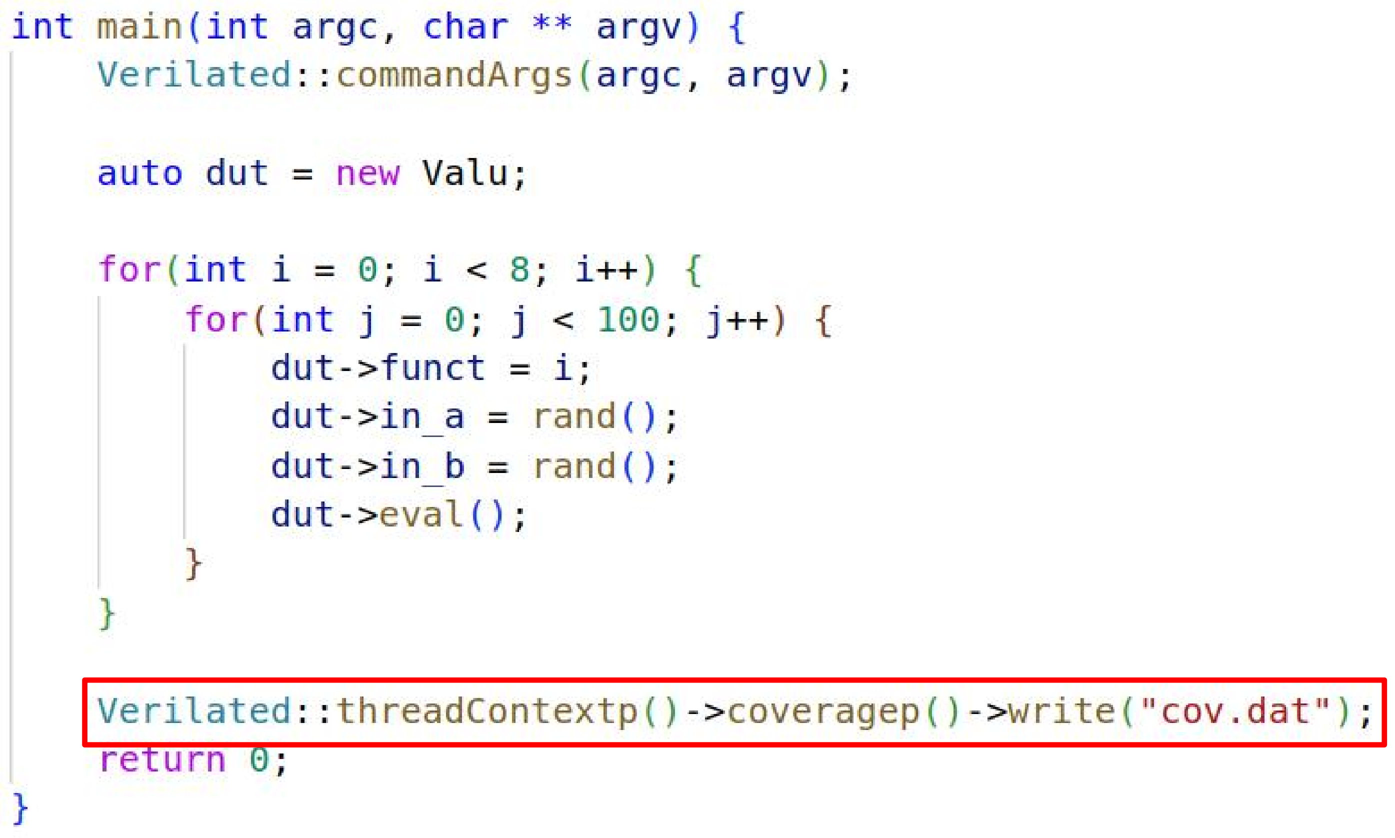


```
cov.dat cov.info &: alu.sv alu_tb.sv tb.cpp
	verilator -cc --build --exe --coverage $^
	./obj_dir/Valu
	verilator_coverage cov.dat -write-info cov.info
```

  - `--coverage`: turn on coverage test
  - `./obj_dir/Valu`: run target program to gen coverage data
  - `verilator_coverage`: convert converage data

Part. 3. Coverage Test

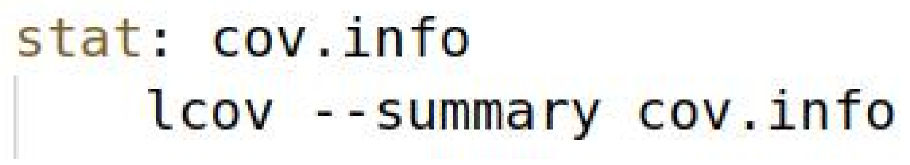
- Add Code to export coverage data:
  - run `make` to do coverage test



```
int main(int argc, char ** argv) {
    Verilated::commandArgs(argc, argv);

    auto dut = new Valu;

    for(int i = 0; i < 8; i++) {
        for(int j = 0; j < 100; j++) {
            dut->funct = i;
            dut->in_a = rand();
            dut->in_b = rand();
            dut->eval();
        }
    }

    Verilated::threadContextp()->coveragep()->write("cov.dat");
    return 0;
}
```

Part. 3. Coverage Test

- show coverage stat: make stat



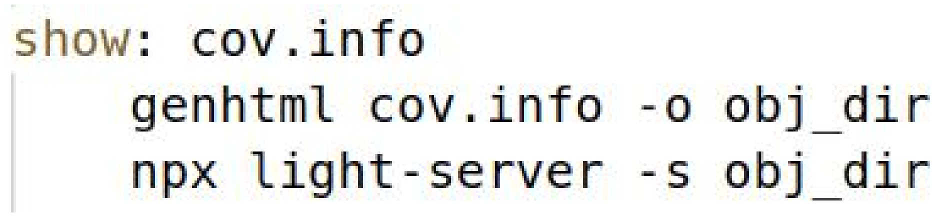
```
stat: cov.info
	lcov --summary cov.info
```

- There is only 97.1% coverage

```
lcov --summary cov.info
Reading tracefile cov.info
Summary coverage rate:
  lines......: 97.1% (67 of 69 lines)
  functions..: no data found
  branches...: no data found
```

Part. 3. Coverage Test

- Open coverage report to find uncovered code

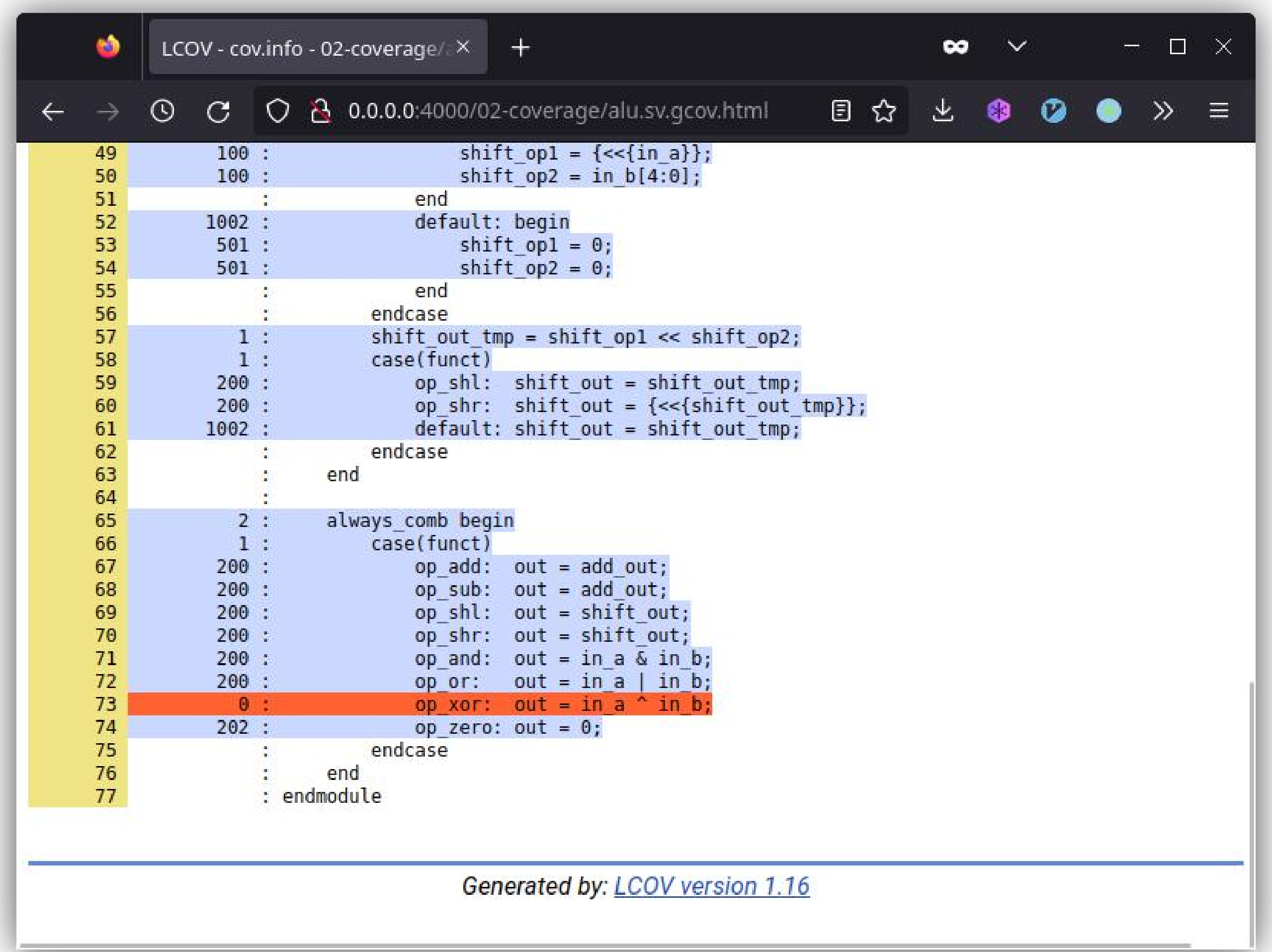


```
show: cov.info
	genhtml cov.info -o obj_dir
	npx light-server -s obj_dir
```

  - `ssh -L 4000:127.0.0.1:4000 root@115.27.161.184 -p xxxx`

- Open your browser, open http://localhost:4000
  - modify the code to cover the

Part. 3. Coverage Test


LCOV - cov.info - 02-coverage/a
0.0.0.0:4000/02-coverage/alu.sv.gcov.html

```
    100 :                 shift_op1 = {<<{in_a}};
    100 :                 shift_op2 = in_b[4:0];
        :             end
   1002 :             default: begin
    501 :                 shift_op1 = 0;
    501 :                 shift_op2 = 0;
        :             end
        :         endcase
      1 :         shift_out_tmp = shift_op1 << shift_op2;
      1 :         case(funct)
    200 :             op_shl:  shift_out = shift_out_tmp;
    200 :             op_shr:  shift_out = {<<{shift_out_tmp}};
   1002 :             default: shift_out = shift_out_tmp;
        :         endcase
        :     end
        :
      2 :     always_comb begin
      1 :         case(funct)
    200 :             op_add:  out = add_out;
    200 :             op_sub:  out = add_out;
    200 :             op_shl:  out = shift_out;
    200 :             op_shr:  out = shift_out;
    200 :             op_and:  out = in_a & in_b;
    200 :             op_or:   out = in_a | in_b;
      0 :             op_xor:  out = in_a ^ in_b;
    202 :             op_zero: out = 0;
        :         endcase
        :     end
        : endmodule
```

Generated by: LCOV version 1.16

Part. 4. Large Design

- In `03-large`, follow the `README.md`