Verilator and C++

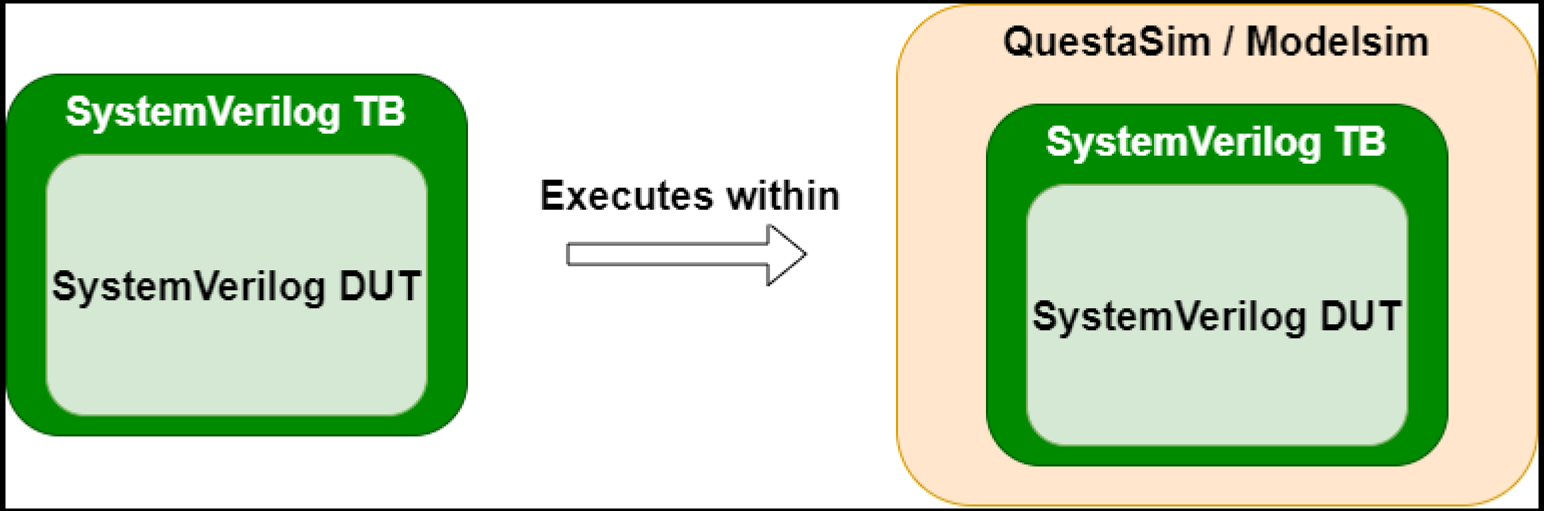
Malcolm McClymont

What is Verilator?

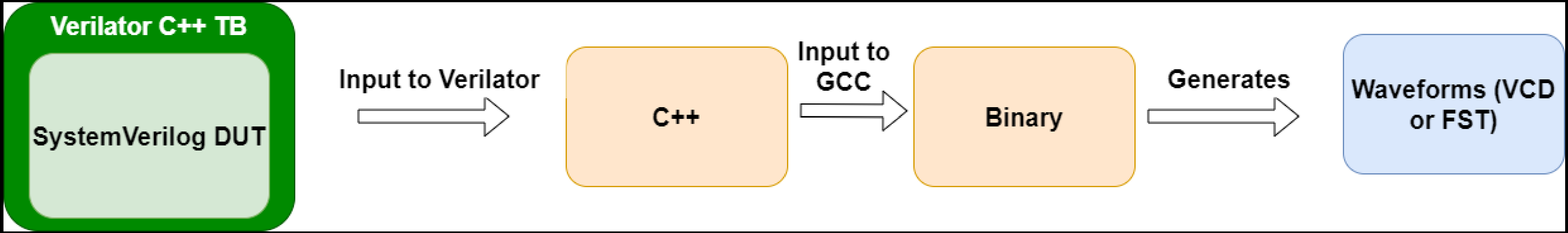
- Verilator is a convertor. It is NOT a simulator
    - Really, C++ is the simulator
    - Verilator allows you to convert Verilog to C++

What is Verilator?

Most SV Simulators

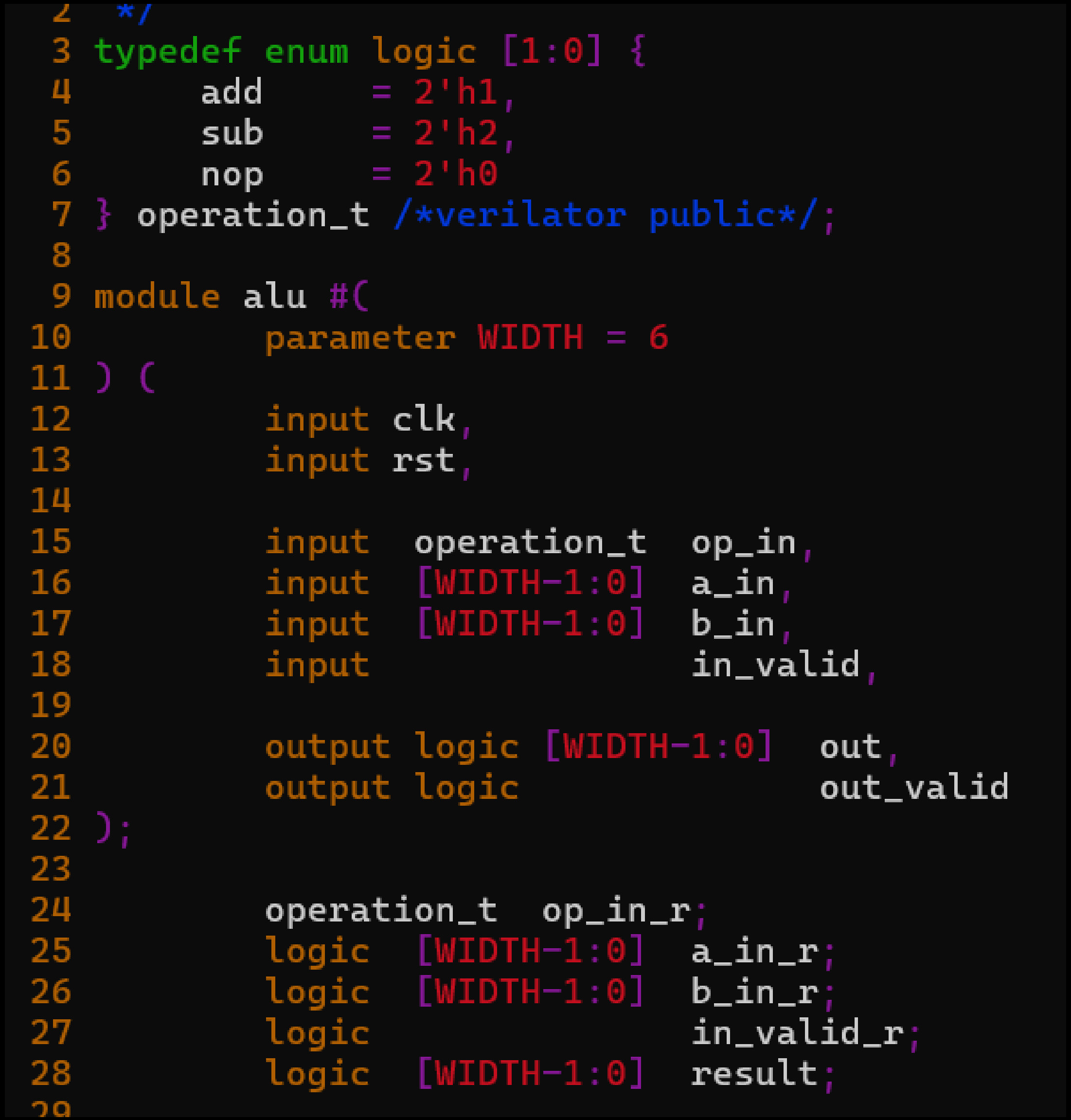


Verilator



Using Verilator

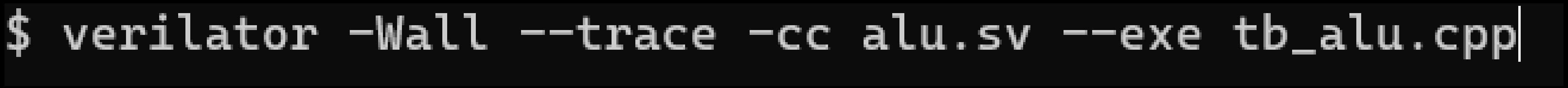
- Using ALU as sample DUT
  - Enums for ADD, SUB, and NOP
    - Add /*Verilator public*/ to access structs in C++
  - 2 FFs between input and output



```
 */
typedef enum logic [1:0] {
     add     = 2'h1,
     sub     = 2'h2,
     nop     = 2'h0
} operation_t /*verilator public*/;

module alu #(
        parameter WIDTH = 6
) (
        input clk,
        input rst,

        input  operation_t  op_in,
        input  [WIDTH-1:0]  a_in,
        input  [WIDTH-1:0]  b_in,
        input               in_valid,

        output logic [WIDTH-1:0]  out,
        output logic              out_valid
);

        operation_t  op_in_r;
        logic  [WIDTH-1:0]  a_in_r;
        logic  [WIDTH-1:0]  b_in_r;
        logic               in_valid_r;
        logic  [WIDTH-1:0]  result;
```

Using Verilator

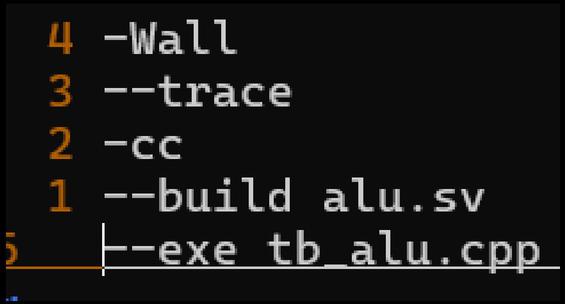
- Verilate a module to compile produce C++
  - -Wall: Enable all warnings
  - -Trace: Enables trace output
  - -cc: Outputs Verilog as C++
  - Alu.sv: The DUT
  - --exe tb_alu.cpp: Include this file as our C++ testbench



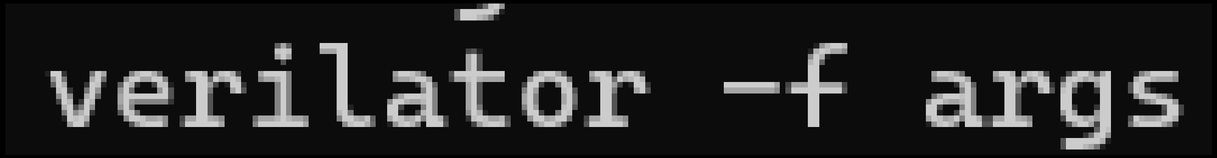
```
$ verilator -Wall --trace -cc alu.sv --exe tb_alu.cpp
```

Using Verilator

- Store arguments in a file or use a Bash script



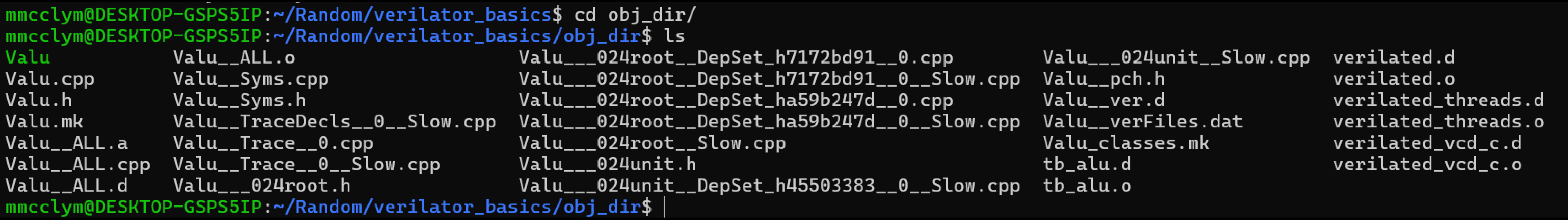
```
-Wall
--trace
-cc
--build alu.sv
--exe tb_alu.cpp
```



```
verilator -f args
```

Using Verilator

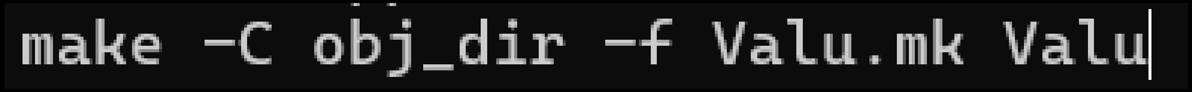
- What did we generate?
  - Valu.cpp: The C++ version of our DUT
  - Valu.mk: Makefile fragment used to create simulation binary
  - Valu__024unit.h: ALU enums (ADD, SUB, NOP)



```
mmcclym@DESKTOP-GSPS5IP:~/Random/verilator_basics$ cd obj_dir/
mmcclym@DESKTOP-GSPS5IP:~/Random/verilator_basics/obj_dir$ ls
Valu          Valu__ALL.o                Valu___024root__DepSet_h7172bd91__0.cpp       Valu___024unit__Slow.cpp  verilated.d
Valu.cpp      Valu__Syms.cpp             Valu___024root__DepSet_h7172bd91__0__Slow.cpp  Valu__pch.h               verilated.o
Valu.h        Valu__Syms.h               Valu___024root__DepSet_ha59b247d__0.cpp       Valu__ver.d               verilated_threads.d
Valu.mk       Valu__TraceDecls__0__Slow.cpp  Valu___024root__DepSet_ha59b247d__0__Slow.cpp  Valu__verFiles.dat        verilated_threads.o
Valu__ALL.a   Valu__Trace__0.cpp         Valu___024root__Slow.cpp                      Valu_classes.mk           verilated_vcd_c.d
Valu__ALL.cpp Valu__Trace__0__Slow.cpp   Valu___024unit.h                              tb_alu.d                  verilated_vcd_c.o
Valu__ALL.d   Valu___024root.h           Valu___024unit__DepSet_h45503383__0__Slow.cpp  tb_alu.o
mmcclym@DESKTOP-GSPS5IP:~/Random/verilator_basics/obj_dir$
```

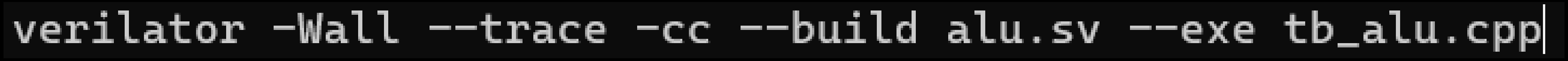
Using Verilator

- Use Makefile fragment to compile sim



```
make -C obj_dir -f Valu.mk Valu
```

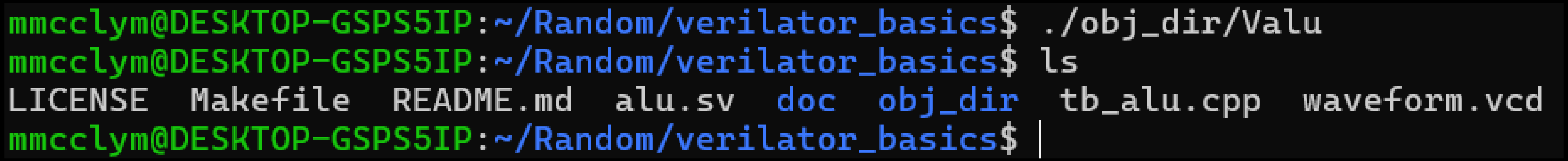
- Or, add the –build flag to Verilator command to automatically call make



```
verilator -Wall --trace -cc --build alu.sv --exe tb_alu.cpp
```

Using Verilator

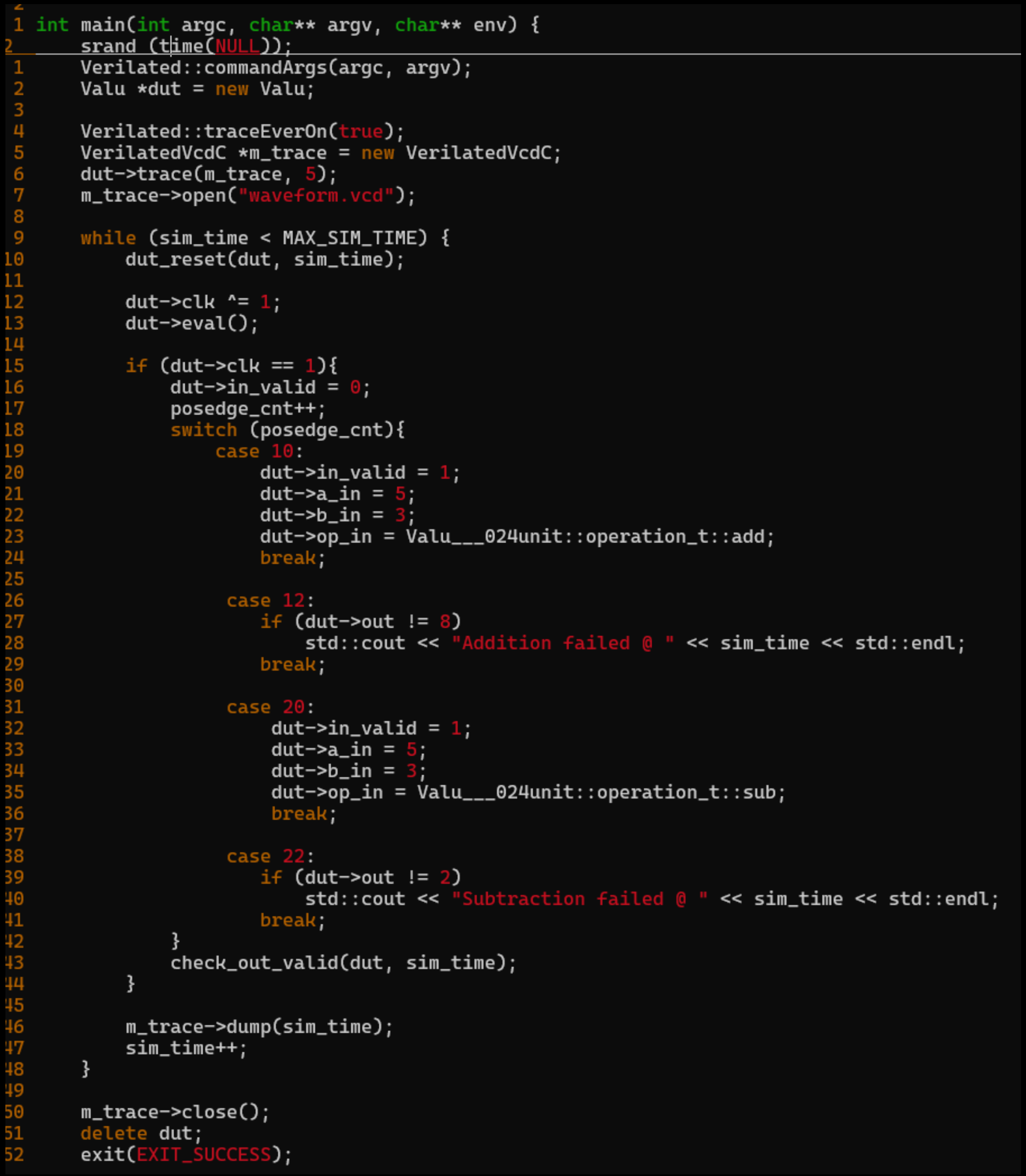
- Finally, call the compiled binary in obj_dir to simulate design
  - Output saved in waveform.vcd



```
mmcclym@DESKTOP-GSPS5IP:~/Random/verilator_basics$ ./obj_dir/Valu
mmcclym@DESKTOP-GSPS5IP:~/Random/verilator_basics$ ls
LICENSE  Makefile  README.md  alu.sv  doc  obj_dir  tb_alu.cpp  waveform.vcd
mmcclym@DESKTOP-GSPS5IP:~/Random/verilator_basics$
```

Writing Testbenches

- Instantiate the DUT
- Run the tests
- Close the DUT



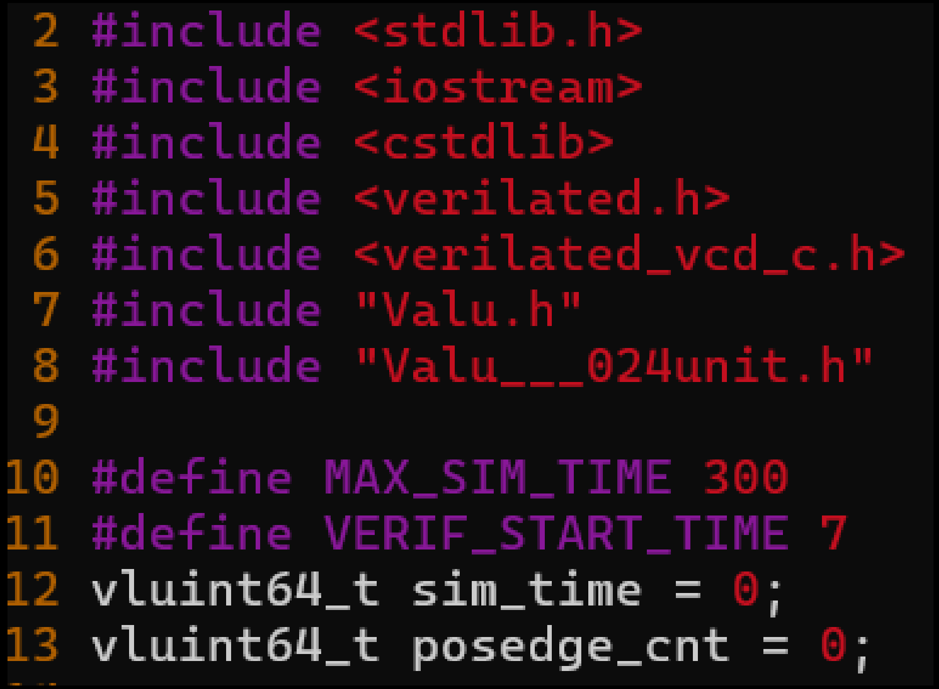
```
int main(int argc, char** argv, char** env) {
    srand (time(NULL));
    Verilated::commandArgs(argc, argv);
    Valu *dut = new Valu;

    Verilated::traceEverOn(true);
    VerilatedVcdC *m_trace = new VerilatedVcdC;
    dut->trace(m_trace, 5);
    m_trace->open("waveform.vcd");

    while (sim_time < MAX_SIM_TIME) {
        dut_reset(dut, sim_time);

        dut->clk ^= 1;
        dut->eval();

        if (dut->clk == 1){
            dut->in_valid = 0;
            posedge_cnt++;
            switch (posedge_cnt){
                case 10:
                    dut->in_valid = 1;
                    dut->a_in = 5;
                    dut->b_in = 3;
                    dut->op_in = Valu___024unit::operation_t::add;
                    break;

                 case 12:
                    if (dut->out != 8)
                        std::cout << "Addition failed @ " << sim_time << std::endl;
                    break;

                 case 20:
                     dut->in_valid = 1;
                     dut->a_in = 5;
                     dut->b_in = 3;
                     dut->op_in = Valu___024unit::operation_t::sub;
                     break;

                 case 22:
                    if (dut->out != 2)
                        std::cout << "Subtraction failed @ " << sim_time << std::endl;
                    break;
            }
            check_out_valid(dut, sim_time);
        }

        m_trace->dump(sim_time);
        sim_time++;
    }

    m_trace->close();
    delete dut;
    exit(EXIT_SUCCESS);
```

Writing Testbenches

- A side note on pointers:
  - Most Verilator objects are handled as pointers
  - Arrow combines dereference and dot
    - Dereference the pointer to get struct, THEN access a field within struct
- Foo->bar is equivalent to (*foo).bar
  - Cannot do foo.bar if foo is a pointer!
    - Would access memory at &foo + struct offset

Writing Testbenches

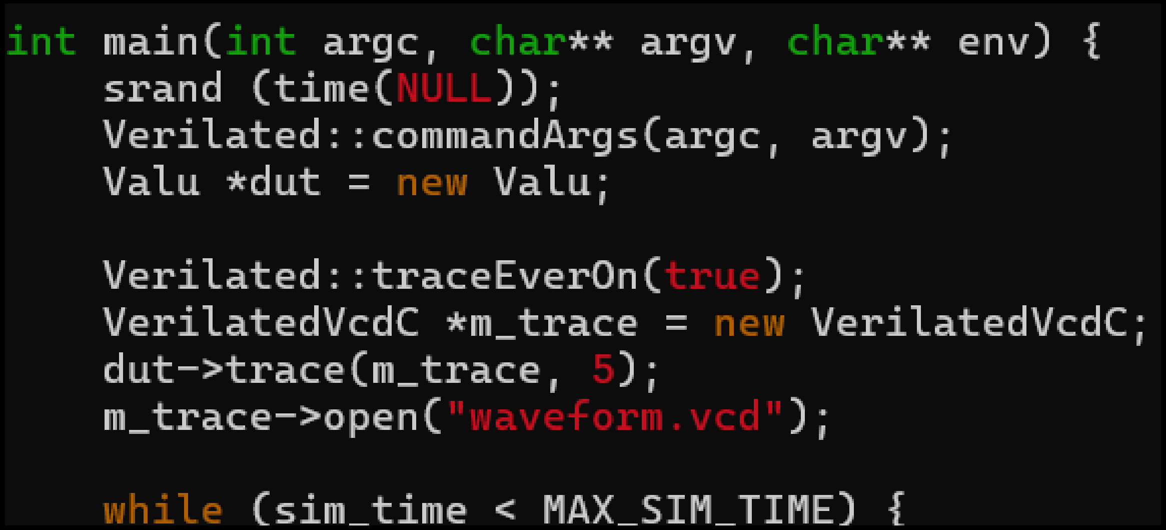
- Verilated.h: Base Verilator functions and data types
- Verilated_vcd_c.h: Enables VCD output
- Valu.h: Our top-level DUT
- Valu__024unit.h: Our DUT enums
- Vluint64_t is typically used for time



```
#include <stdlib.h>
#include <iostream>
#include <cstdlib>
#include <verilated.h>
#include <verilated_vcd_c.h>
#include "Valu.h"
#include "Valu___024unit.h"

#define MAX_SIM_TIME 300
#define VERIF_START_TIME 7
vluint64_t sim_time = 0;
vluint64_t posedge_cnt = 0;
```

Writing Testbenches

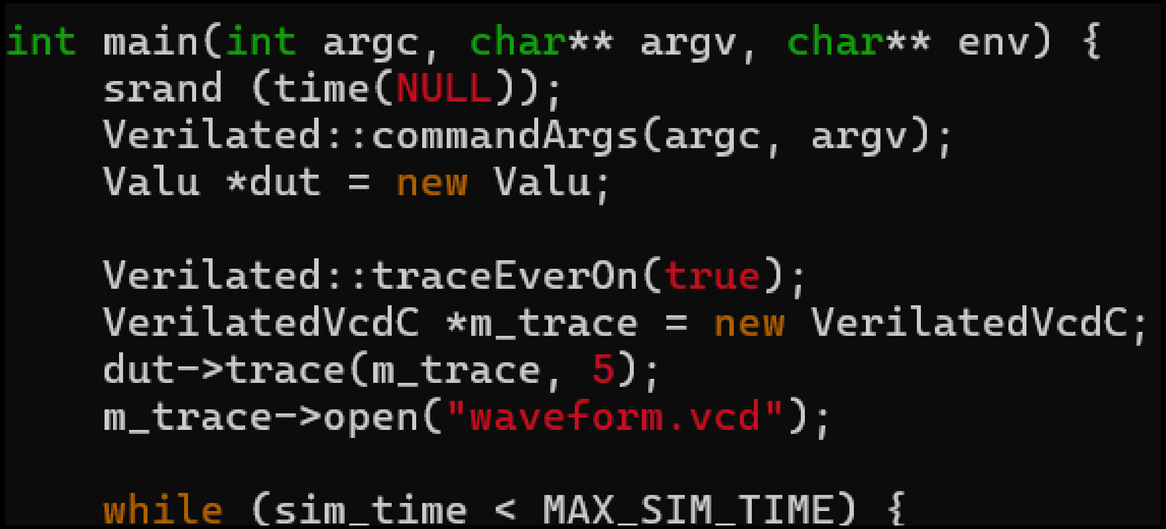
- Instantiate the DUT
- commandArgs: Allows you to pass arguments to Verilator executable ([more info here]())
- Valu* dut: Instantiates an instance of DUT
- traceEverOn(true): Enables waveform generation



```
int main(int argc, char** argv, char** env) {
    srand (time(NULL));
    Verilated::commandArgs(argc, argv);
    Valu *dut = new Valu;

    Verilated::traceEverOn(true);
    VerilatedVcdC *m_trace = new VerilatedVcdC;
    dut->trace(m_trace, 5);
    m_trace->open("waveform.vcd");

    while (sim_time < MAX_SIM_TIME) {
```

Writing Testbenches

- VerilatedVcc *m_trace: Creates waveform object
- Dut->trace(m_trace,5): Assigns waveform object to DUT and sets max depth of waves to 5
- m_trace->open(): Like fopen in C. Dump its contents each cycle

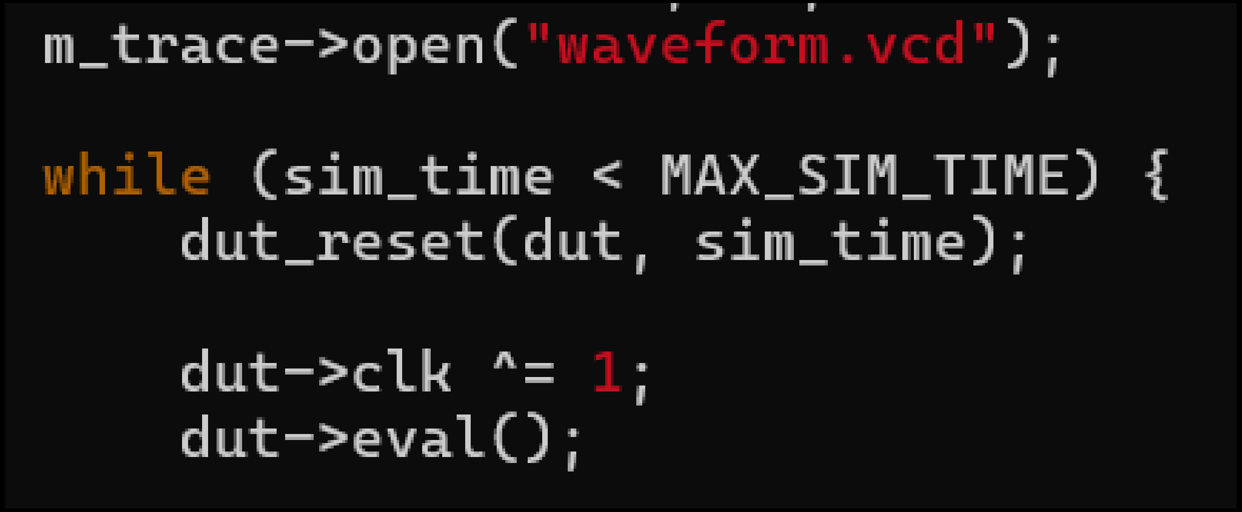


```
int main(int argc, char** argv, char** env) {
    srand (time(NULL));
    Verilated::commandArgs(argc, argv);
    Valu *dut = new Valu;

    Verilated::traceEverOn(true);
    VerilatedVcdC *m_trace = new VerilatedVcdC;
    dut->trace(m_trace, 5);
    m_trace->open("waveform.vcd");

    while (sim_time < MAX_SIM_TIME) {
```

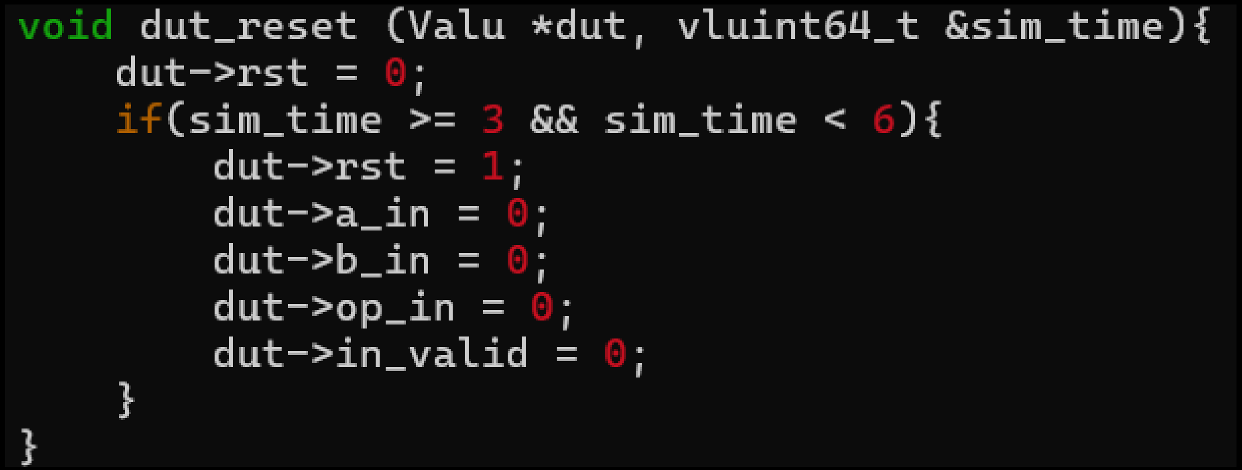
Writing Testbenches

- Reset the DUT
  - Use arrow notation to set signal values
  - Call function each simulation step for multi-cycle reset

  - Dut->clk ^= 1: Set clock to 0 or 1
  - Dut->eval: Evaluate all signals in DUT
    - Be mindful of where you call this!



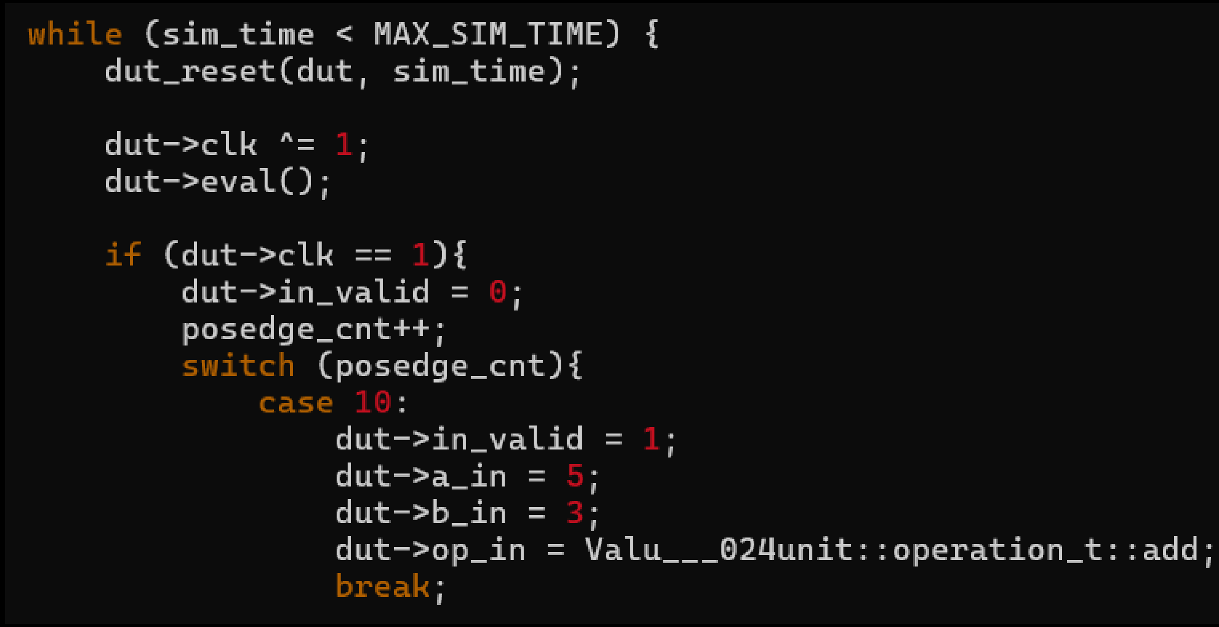
```
m_trace->open("waveform.vcd");

while (sim_time < MAX_SIM_TIME) {
    dut_reset(dut, sim_time);

    dut->clk ^= 1;
    dut->eval();
```



```
void dut_reset (Valu *dut, vluint64_t &sim_time){
    dut->rst = 0;
    if(sim_time >= 3 && sim_time < 6){
        dut->rst = 1;
        dut->a_in = 0;
        dut->b_in = 0;
        dut->op_in = 0;
        dut->in_valid = 0;
    }
}
```

Writing Testbenches

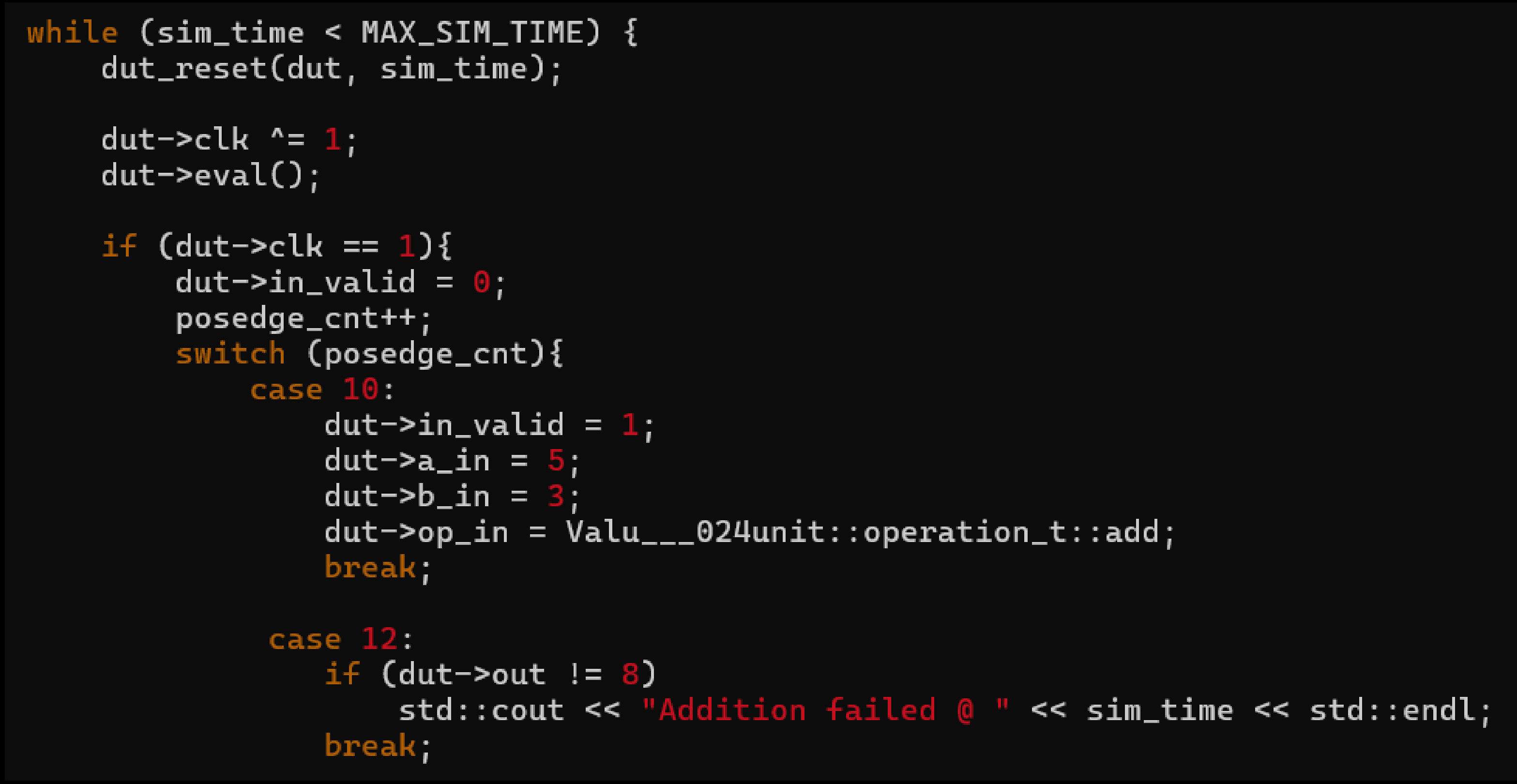
- Add counter for positive clock edges
- Switch(posedge_cnt): Do tests on certain rising clock edges
  - For example: Add 2 numbers



```
while (sim_time < MAX_SIM_TIME) {
    dut_reset(dut, sim_time);

    dut->clk ^= 1;
    dut->eval();

    if (dut->clk == 1){
        dut->in_valid = 0;
        posedge_cnt++;
        switch (posedge_cnt){
            case 10:
                dut->in_valid = 1;
                dut->a_in = 5;
                dut->b_in = 3;
                dut->op_in = Valu___024unit::operation_t::add;
                break;
```

Writing Testbenches

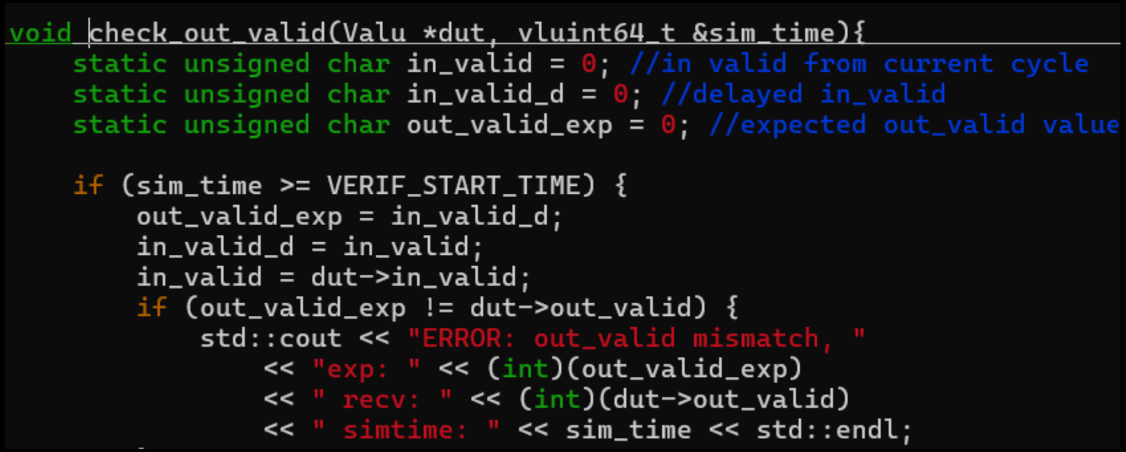
- Read DUT output with arrow notation
- Use cout to print
  - Can be used like assertions (but won’t stop your simulation)



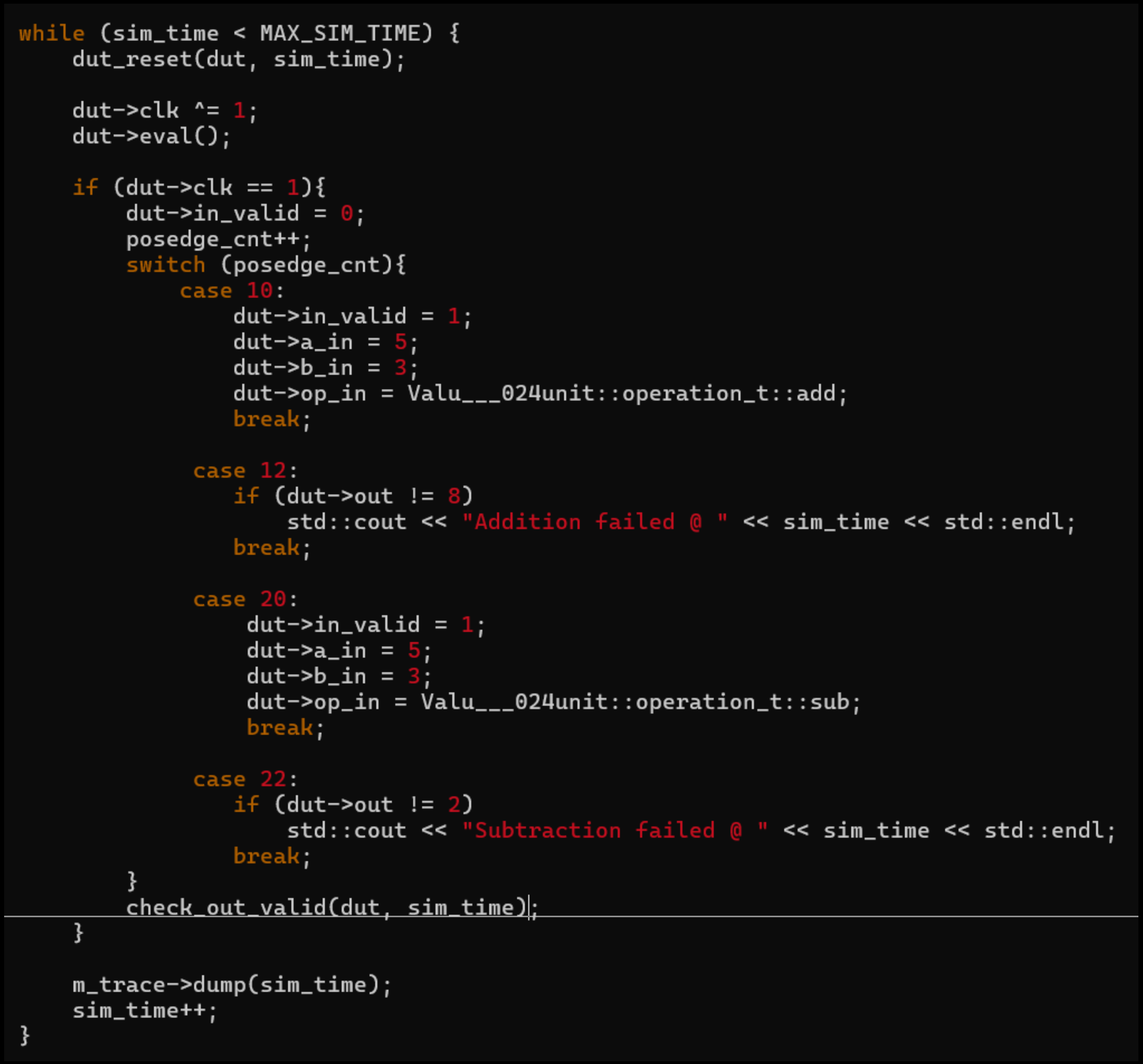
```
while (sim_time < MAX_SIM_TIME) {
  dut_reset(dut, sim_time);

  dut->clk ^= 1;
  dut->eval();

  if (dut->clk == 1){
    dut->in_valid = 0;
    posedge_cnt++;
    switch (posedge_cnt){
      case 10:
        dut->in_valid = 1;
        dut->a_in = 5;
        dut->b_in = 3;
        dut->op_in = Valu___024unit::operation_t::add;
        break;

       case 12:
        if (dut->out != 8)
          std::cout << "Addition failed @ " << sim_time << std::endl;
        break;
```

Writing Testbenches

- Can add arbitrary number of test cases
- Can define functions to test DUT
- Don’t forget to dump trace and increment sim_time



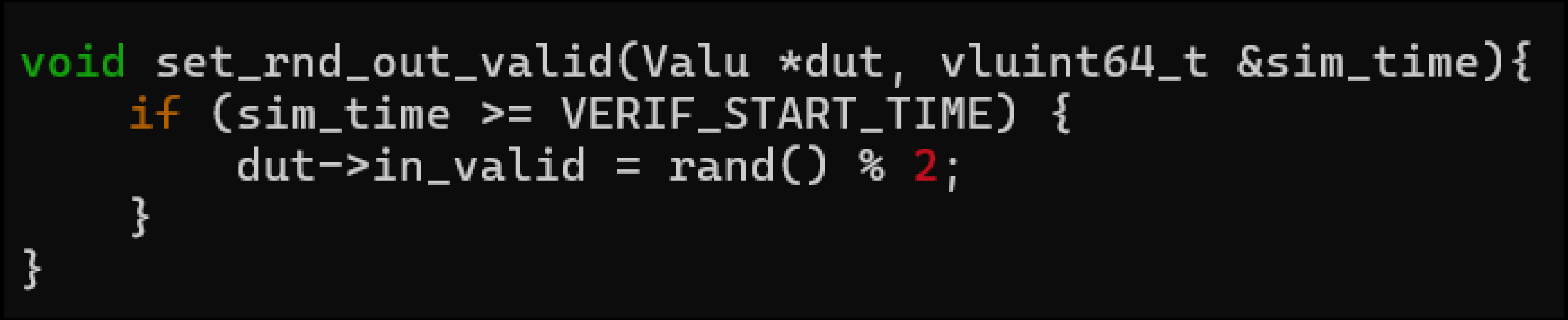
```
void check_out_valid(Valu *dut, vluint64_t &sim_time){
    static unsigned char in_valid = 0; //in valid from current cycle
    static unsigned char in_valid_d = 0; //delayed in_valid
    static unsigned char out_valid_exp = 0; //expected out_valid value

    if (sim_time >= VERIF_START_TIME) {
        out_valid_exp = in_valid_d;
        in_valid_d = in_valid;
        in_valid = dut->in_valid;
        if (out_valid_exp != dut->out_valid) {
            std::cout << "ERROR: out_valid mismatch, "
                << "exp: " << (int)(out_valid_exp)
                << " recv: " << (int)(dut->out_valid)
                << " simtime: " << sim_time << std::endl;
```



```
while (sim_time < MAX_SIM_TIME) {
    dut_reset(dut, sim_time);

    dut->clk ^= 1;
    dut->eval();

    if (dut->clk == 1){
        dut->in_valid = 0;
        posedge_cnt++;
        switch (posedge_cnt){
            case 10:
                dut->in_valid = 1;
                dut->a_in = 5;
                dut->b_in = 3;
                dut->op_in = Valu___024unit::operation_t::add;
                break;

             case 12:
                if (dut->out != 8)
                    std::cout << "Addition failed @ " << sim_time << std::endl;
                break;

             case 20:
                 dut->in_valid = 1;
                 dut->a_in = 5;
                 dut->b_in = 3;
                 dut->op_in = Valu___024unit::operation_t::sub;
                 break;

             case 22:
                if (dut->out != 2)
                    std::cout << "Subtraction failed @ " << sim_time << std::endl;
                break;
        }
        check_out_valid(dut, sim_time);
    }

    m_trace->dump(sim_time);
    sim_time++;
}
```

Writing Testbenches

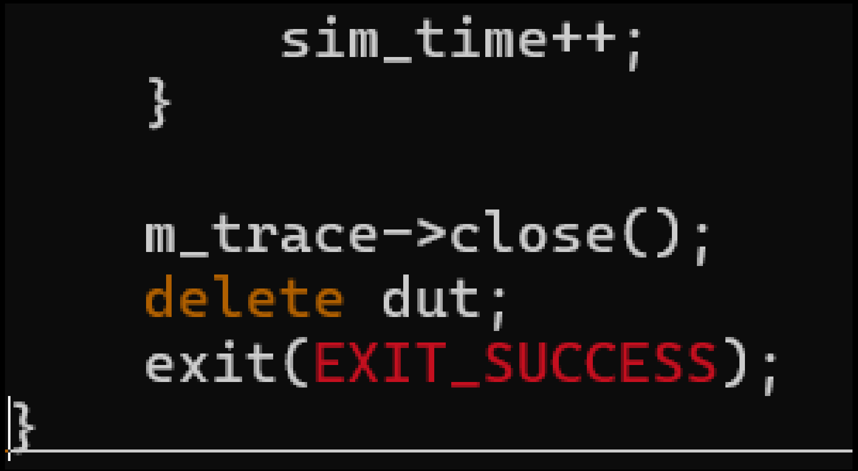
- Can do testing with randomized inputs
  - Very useful for coverage
  - Used by industry in combination with unit tests



```
void set_rnd_out_valid(Valu *dut, vluint64_t &sim_time){
    if (sim_time >= VERIF_START_TIME) {
        dut->in_valid = rand() % 2;
    }
}
```

Writing Testbenches

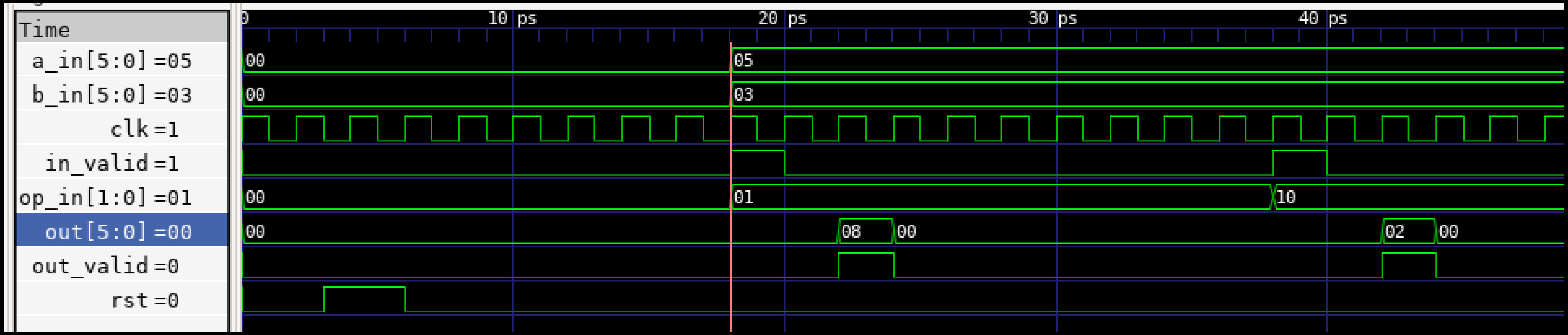
- Finish simulation by closing trace file and freeing memory



```
        sim_time++;
    }

    m_trace->close();
    delete dut;
    exit(EXIT_SUCCESS);
}
```

Testbench Output

- Open waveform output with any waveform viewer (typically GTKWave)



Advantages of Verilator

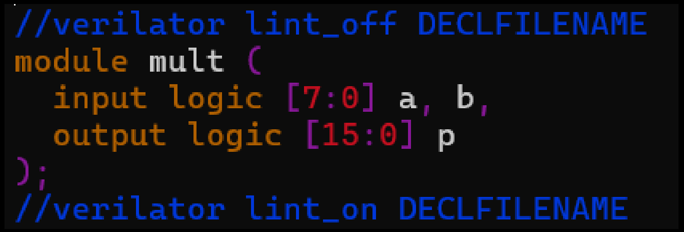
- Can easily use software as a golden model
- Access to all C/C++ features and libraries
- Simulation is multi-threaded by default, runs faster than Questa/Modelsim
- Very descriptive warnings

Disadvantages of Verilator

- Access to all C/C++ “features”
  - Still can’t dereference null pointers
- Currently cannot support 4-state logic (no X or Z values)

General Verilator Tips

- Add "–j 16" to verilate command for multi-threaded compilation
  - -j 0 uses all cores, but be mindful of load on asicfab
- Optimize your design, NOT your testbench!
  - Prioritize testbench correctness and flexibility over speed
- Stick to one dut->eval() per sim_time cycle, unless you need something specific
- Use lint directives to disable/enable certain warnings in your Verilog



```
//verilator lint_off DECLFILENAME
module mult (
  input logic [7:0] a, b,
  output logic [15:0] p
);
//verilator lint_on DECLFILENAME
```

Resources

- https://itsembedded.com/dhd_list/
- https://www.veripool.org/ftp/verilator_doc.pdf
- https://verilator.org/guide/latest/index.html
- https://verilator.org/guide/latest/warnings.html