Verilog: A Practice Approach

junyu33

official website

- https://www.veripool.org/verilator/
- https://verilator.org

installation

- verilator

```
sudo apt-get install verilator   # On Debian or Ubuntu
```

- gtkwave

```
sudo apt-get install gtkwave   # On Debian or Ubuntu
```

hello world

```
mkdir playground
touch top.v main.cpp
```

- top.v

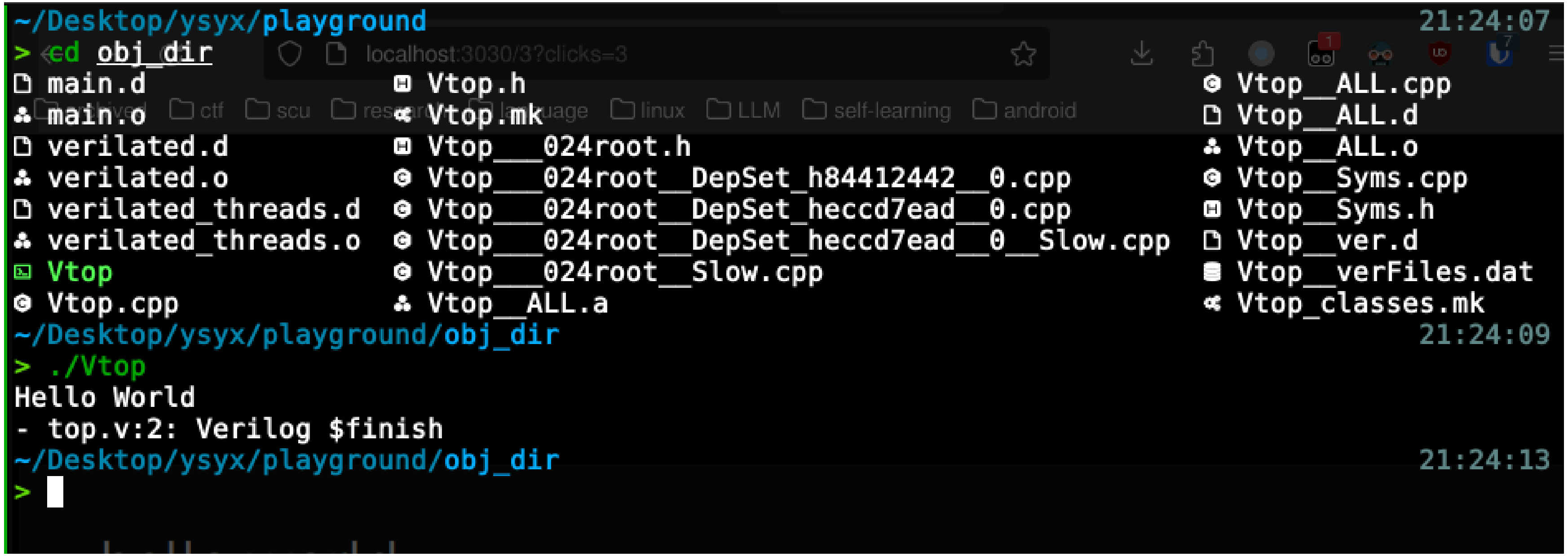
```
module top;
  initial begin $display("Hello World"); $finish; end
endmodule
```

- main.cpp

```
#include "Vtop.h"
#include "verilated.h"
int main(int argc, char** argv) {
  VerilatedContext* contextp = new VerilatedContext;
  contextp->commandArgs(argc, argv);
  Vtop* top = new Vour{contextp};
  while (!contextp->gotFinish()) { top->eval(); }
  delete top;
  delete contextp;
  return 0;
}
```

first run

```
verilator --cc --exe --build -j 0 -Wall main.cpp top.v
```



```
~/Desktop/ysyx/playground                                                        21:24:07
> cd obj_dir
main.d                     Vtop.h                                    Vtop__ALL.cpp
main.o                     Vtop.mk                                   Vtop__ALL.d
verilated.d                Vtop___024root.h                          Vtop__ALL.o
verilated.o                Vtop___024root__DepSet_h84412442__0.cpp   Vtop__Syms.cpp
verilated_threads.d        Vtop___024root__DepSet_heccd7ead__0.cpp   Vtop__Syms.h
verilated_threads.o        Vtop___024root__DepSet_heccd7ead__0__Slow.cpp Vtop__ver.d
Vtop                       Vtop___024root__Slow.cpp                  Vtop__verFiles.dat
Vtop.cpp                   Vtop__ALL.a                               Vtop_classes.mk
~/Desktop/ysyx/playground/obj_dir                                                21:24:09
> ./Vtop
Hello World
- top.v:2: Verilog $finish
~/Desktop/ysyx/playground/obj_dir                                                21:24:13
> 
```

and gate

- top.v

```
module top(
  input a,
  input b,
  output c
);
  assign c = a & b;
endmodule
```

- main.cpp

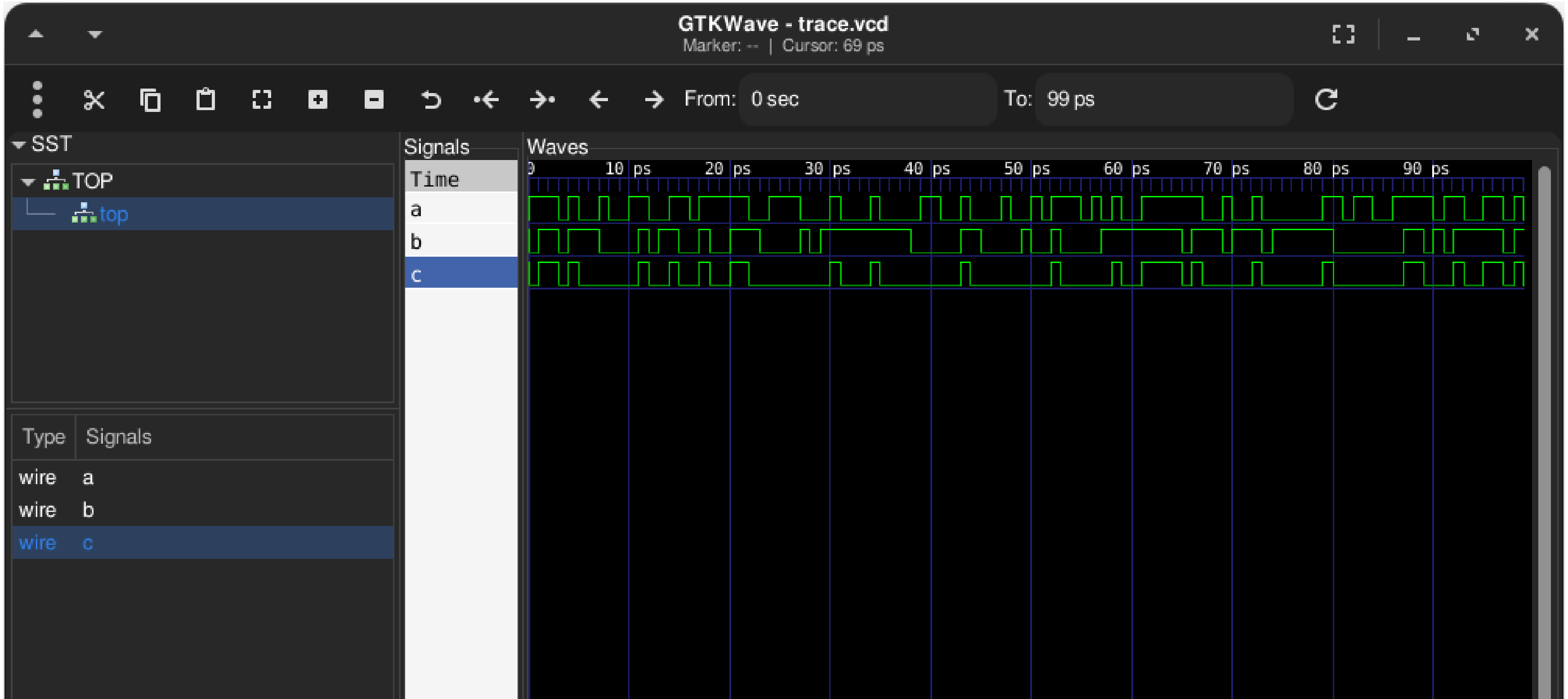
```
#include "Vtop.h"
#include "verilated.h"
#include "verilated_vcd_c.h"
#define TIME_LIMIT 100
int main(int argc, char** argv) {
  VerilatedContext* contextp = new VerilatedContext;
  contextp->commandArgs(argc, argv);
  Vtop* top = new Vtop{contextp};
  VerilatedVcdC* tfp = new VerilatedVcdC; // class for tackle vcd
  Verilated::traceEverOn(true);
  top->trace(tfp, TIME_LIMIT - 1);        // trace the signal
  tfp->open("trace.vcd");
  while (contextp->time() < TIME_LIMIT && !Verilated::gotFinish()) {
    /* your variable here */
    top->a = rand() % 2; // generate input
    top->b = rand() % 2;
    top->eval();          // run the circuit
    tfp->dump(contextp->time()); // save the result
    contextp->timeInc(1); // increase a clock cycle
  }

  tfp->close();
  delete top;
  delete contextp;
  return 0;
}
```

- Makefile

```
clean:
  rm -rf obj_dir
build:
  verilator --cc --exe --trace --build -j 0 -Wall main.cpp top.v
run: build
  ./obj_dir/Vtop
```

second run (with signals)

```
make run
gtkwave ./trace.vcd
```



Type	Signals
wire	a
wire	b
wire	c

full adder

- top.v

```
module top(
  input a, b, carry_in
  output sum, carry_out
);
  assign sum = a ^ b ^ carry_in;
  assign carry_out = (a & b) | (a & carry_in) | (b & carry_in);
endmodule
```

OR

```
module top(
  input a, b, carry_in
  output sum, carry_out
);
  assign {carry_out, sum} = a + b + carry_in; // {} and ',' mean concat
endmodule
```

The former is more like structural modeling, while dataflow modeling for the latter.

multi-bit addr

```
module full_addr(
  input a, b, carry_in,
  output sum, carry_out
);
  assign sum = a ^ b ^ carry_in;
  assign carry_out = (a & b) | (a & carry_in) | (b & carry_in);
endmodule

module top(
  input [7:0] a, b,
  input carry_in,
  output [7:0] sum,
  output carry_out
);
  wire [6:0] carry_tmp;
  full_addr fa0(a[0], b[0], carry_in, sum[0], carry_tmp[0]);
  full_addr fa1(a[1], b[1], carry_tmp[0], sum[1], carry_tmp[1]);
  full_addr fa2(a[2], b[2], carry_tmp[1], sum[2], carry_tmp[2]);
  full_addr fa3(a[3], b[3], carry_tmp[2], sum[3], carry_tmp[3]);
  full_addr fa4(a[4], b[4], carry_tmp[3], sum[4], carry_tmp[4]);
  full_addr fa5(a[5], b[5], carry_tmp[4], sum[5], carry_tmp[5]);
  full_addr fa6(a[6], b[6], carry_tmp[5], sum[6], carry_tmp[6]);
  full_addr fa7(a[7], b[7], carry_tmp[6], sum[7], carry_out);
endmodule
```

- dataflow modeling

```
module top(
  input [7:0] a, b,
  input carry_in,
  output [7:0] sum,
  output carry_out
);
  assign {carry_out, sum} = a + b + {7'b0, carry_in};
endmodule
```

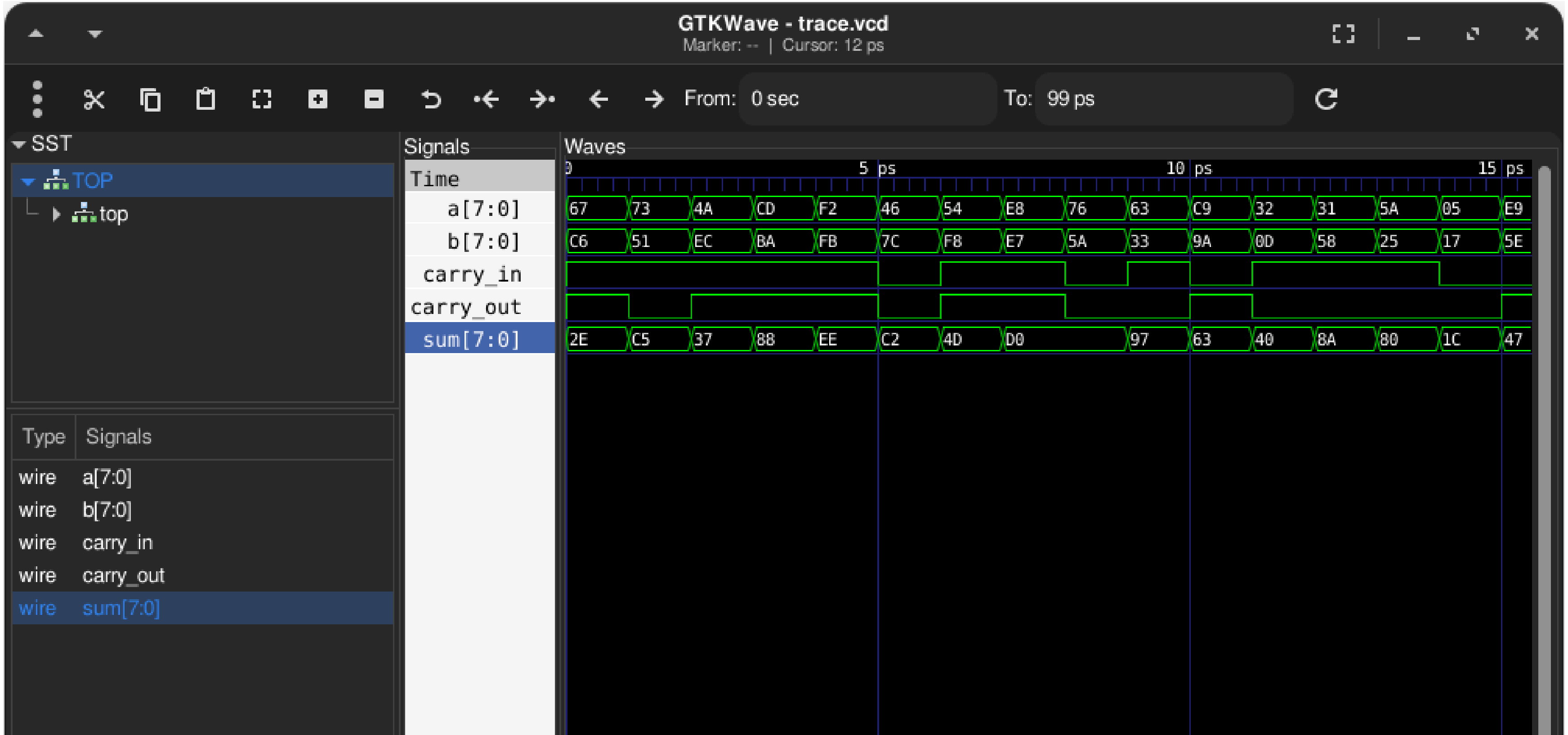
to simplify the procedure, I'll use dataflow modeling more to save the space

- main.cpp

```
while (contextp->time() < TIME_LIMIT && !Verilated::gotFinish()) {
  /* your variable here */
  top->a = rand() % 256; // can directly tranfer an integer to signals
  top->b = rand() % 256;
  top->carry_in = rand() % 2;
  top->eval();
  tfp->dump(contextp->time());
  contextp->timeInc(1);
}
```

3rd run

- 0x67 + 0xc6 + 1 = 0x12e



GTKWave - trace.vcd
Marker: -- | Cursor: 12 ps

From: 0 sec To: 99 ps

SST
TOP
top

Type	Signals
wire	a[7:0]
wire	b[7:0]
wire	carry_in
wire	carry_out
wire	sum[7:0]

```
Signals      Waves
Time         0                   5 ps                10 ps               15 ps
a[7:0]       67 73 4A CD F2 46 54 E8 76 63 C9 32 31 5A 05 E9
b[7:0]       C6 51 EC BA FB 7C F8 E7 5A 33 9A 0D 58 25 17 5E
carry_in
carry_out
sum[7:0]     2E C5 37 88 EE C2 4D D0 97 63 40 8A 80 1C 47
```

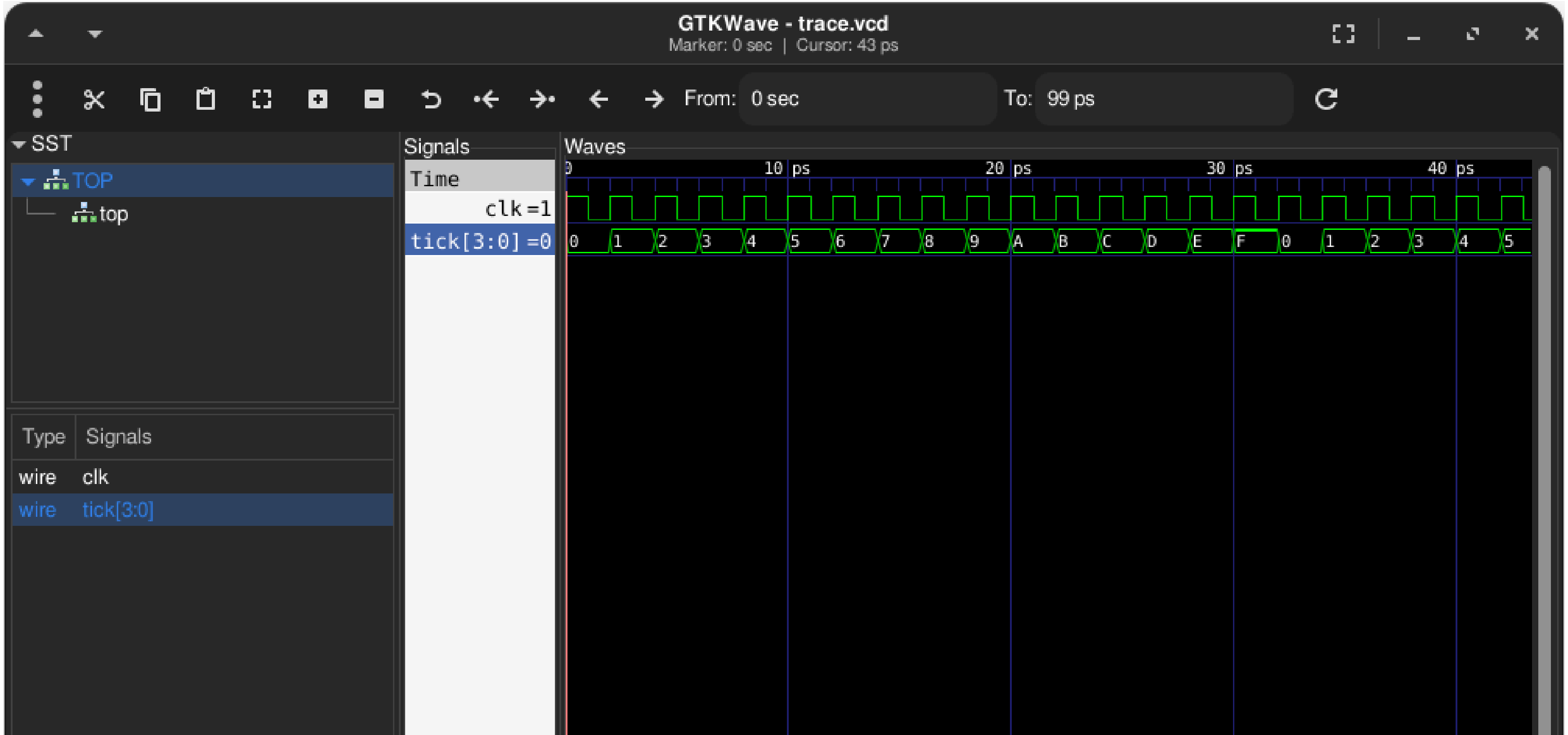
counter

```
module top(
  input clk,
  output [3:0] tick
);
  reg [3:0] saved_tick; // reg can keep its state during different clocks
  always @(posedge clk) begin
    // notice the symbol "<=", it is like a delayed assignment by one clock cycle
    saved_tick <= saved_tick + 1;
  end
  assign tick = saved_tick;
endmodule
```

```
while (contextp->time() < TIME_LIMIT && !Verilated::gotFinish()) {
  top->clk = 1; top->eval(); // because it is initially 1, so no posedge
  tfp->dump(contextp->time());
  contextp->timeInc(1);

  top->clk = 0; top->eval();
  tfp->dump(contextp->time());
  contextp->timeInc(1);
  // two clock cycles later, the counter will increase 1
}
```

4th run

- as you can see, after 32 clock cycles, it returned to initial state



OK, we used four pieces of code to reviewed "digital circuit design" course.

So, lets do something more interesting!!!

instruction cycle

- Let's implement a finite-state machine with one 2-bit reg, with totally 4 instructions!

```
int pseudo_mem[16] = {
  // 0b00: mem = 0
  // 0b01: mem = 1
  // 0b10: mem = mem - 1
  // 0b11: mem = mem + 1
  1, 3, 0, 2
};
```

```
while (contextp->time() < TIME_LIMIT && !Verilated::gotFinish()) {
  /* your variable here */
  top->clk = 0;
  top->inst = pseudo_mem[contextp->time() / 2] % 4; // FETCH
  top->eval();
  tfp->dump(contextp->time());
  contextp->timeInc(1);

  top->clk = 1;
  top->eval();
  tfp->dump(contextp->time());
  contextp->timeInc(1);
}
```

- Here is the implement of reading/writing a register

```verilog
module mem(
  input clk,
  input [1:0] data_in,
  output[1:0] data_out
);
  reg [1:0] mem;    // acutual register like  rax,rbx,...  etc. in AMD64
  assign data_out = mem;
  always @(posedge clk) begin
    mem <= data_in;
  end
endmodule
```

- To simplify, we suppose every instruction we need to write something into the register, so we omit the `write_enable` signal.
- other modules can always read `mem` from `data_out`, but can only write into register when a positive edge of clock occurs.

- Then let's look at the ALU (arithmetic logic unit) part

```
module alu(
  input op,
  input [1:0] data_in,
  output[1:0] data_out
);
  assign data_out = op ? data_in + 1 : data_in - 1;
endmodule
```

- Mostly the ALU part is designed by combinational logic circuit.
- If you still remeber the opcode I previously provided, you can find that `0b10` and `0b11` correspond with the content in ALU.

```
  // 0b00: mem = 0
  // 0b01: mem = 1
  // 0b10: mem = mem - 1
  // 0b11: mem = mem + 1
```

- Before going to the next page, can you try to figure out what the top module will be like?

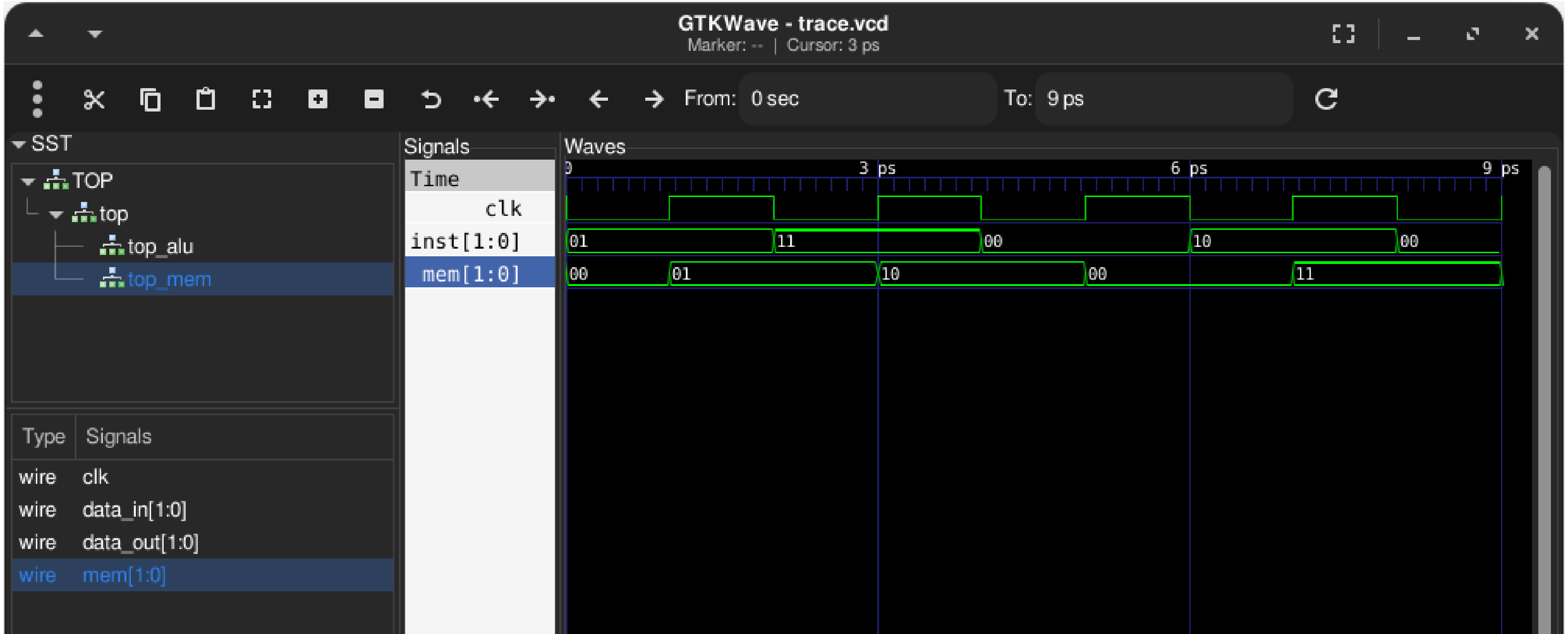
- OK, let me show the possible answer:

```verilog
module top(
  input clk,
  input[1:0] inst
);
  wire [1:0] alu_out, pre_reg, post_reg;

  // READ && WRITE BACK
  mem top_mem(clk, post_reg, pre_reg);
  // EXECUTE
  alu top_alu(inst[0], pre_reg, alu_out);
  // DECODE
  assign post_reg = inst[1] ? alu_out : {1'b0, inst[0]};
endmodule
```

- first, the `top` module FETCH the instruction from `main.cpp`.
- second, read the reg value from `top_mem`.
- third, DECODE `inst[1]` to decide whether the opcode needs calculation.
- fourth, EXECUTE (i.e. posedge) to write the final value back into reg.

5th run

```
mem = 1  // 1
mem += 1 // 2
mem = 0  // 0
mem -= 1 // 3
```


GTKWave - trace.vcd
Marker: -- | Cursor: 3 ps

From: 0 sec To: 9 ps

SST
TOP
top
top_alu
top_mem

Type	Signals
wire	clk
wire	data_in[1:0]
wire	data_out[1:0]
wire	mem[1:0]

Signals
Time
clk
inst[1:0]
mem[1:0]

Waves
0 3 ps 6 ps 9 ps
inst: 01 11 00 10 00
mem: 00 01 10 00 11

final challenge: int64 to double

- i.e. `cvtsi2sd` in x86 asm
- IEEE 754 standard: 1-bit sign, 11-bit exponent, 52-bit mantissa
- e.g.: $(100111001)_2$'s exponent is 8,
- first, it is a positive integer, so the sign-bit is `0`,
- we add 8 into 1023 to make the exponent bits,
- cut off the MSB and get the mantissa.

So, the final result is:

`0 10000000111 001110010000 … 0`

but... we still have a problem,

rounding

supoose we have $(11110000111100001111000011110000111100001111000010101010)_2$

after counting for 53 bits, we find:

`11110000111100001111000011110000111100001111000010101` are accurate but the last `010` bits are rounded.

according to the round rule, $(0.010)_2$ is smaller then $(0.1)_2$, so we round down this.

but what will happen if the omitted bits just euqal to $(0.1)_2$?

Please guess a little bit.

The answer is Banker’s Rounding: Round to nearest, ties to even.

Do you remember the first lecture in 大学物理实验 course？ That’s it.

Banker's Rounding

- Suppose we have $(1111000011110000111100001111000011110000111100001111000010100100)_2$, the accurate bits are `1111000011110000111100001111000011110000111100001111000010100` and omitted bits are `100`, the latter is just equal to $(0.1)_2$.
- if we round it up, the LSB will be `1`, otherwise the LSB will be `0`, so we should round it down.
- Another example, we have $(1111000011110000111100001111000011110000111100001111000010101100)_2$, the accurate bits are `1111000011110000111100001111000011110000111100001111000010101` and omitted bits are still `100`.
- if we round it up, the LSB will be `0`, otherwise the LSB will be `1`, so we should round it up.
- That's it. Let's take a look at its implementation.

implementation

- calc_significant_bit

```
module calc_significant_bit (
  input [63:0] a,
  output [5:0] b
);
  always @* begin
    begin
      for (int i = 63; i ⩾ 0; i--) begin
        if (a[i]) begin
          b = i[5:0];
          break;
        end
      end
    end
  end
endmodule
```

- This is a typical example of **behavioral modeling**, it is easy for us to understand the procedure. But it is hard for us to debug, and for EDA tools to synthesize. What's more, the performance will drop, **especially when you want to write every complex module in a programmer's way, not thinking of we're acutally building the hardware.**

- calc_mantissa

```
module mantissa (
  input [63:0] a,
  input [5:0] exp,
  output [51:0] b
);
  always @* begin
    if (exp <= 52) begin
      b[51:51-exp+1] = a[exp-1:0];
    end else begin
      b[51:0] = a[exp-1:exp-52];
      // Banker's rounding
      if (a[exp-53] == 1 && (exp == 53 || a[exp-54:0] == 0)) begin
        b = b + {51'b0, b[0]};
      end else begin
        b = b + {51'b0, a[exp-53]};
      end
    end
  end
endmodule
```

- Of course, verilog itself doesn’t support slicing with dynamic variables, can you come up with the solution?

- `top`

```verilog
module top (
  input [63:0] a,
  output [63:0] b
);
  wire sign;
  wire [5:0] exp;
  wire [10:0] fixed_exp;
  wire [51:0] mant;
  wire [63:0] result;
  calc_significant_bit calc_significant_bit_0 (a, exp);
  mantissa mantissa_0 (a, exp, mant);
  assign sign = a[63];
  assign fixed_exp = {5'b0, exp} + 1023;
  assign result = {sign, fixed_exp, mant};
  assign b = result;
endmodule
```

- how to transfer `int64` to double numbers from memory directly without conversion?
- Hint: `union`
- One last question, this pseudocode only handles positive integers, what about negatives?