Verilog Tutorial

Outline

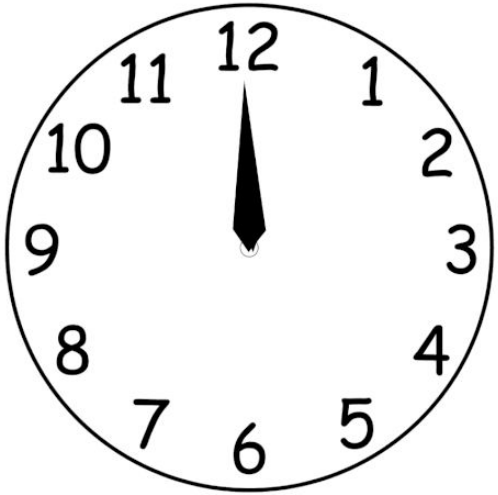
- Digital logic (Digital Systems Design and Laboratory)
- Verilog introduction

Digital logic

- What is digital logic?
- Boolean algebra
- Logic gates
- Combinational logic
- Sequential logic

What is digital?

Analog

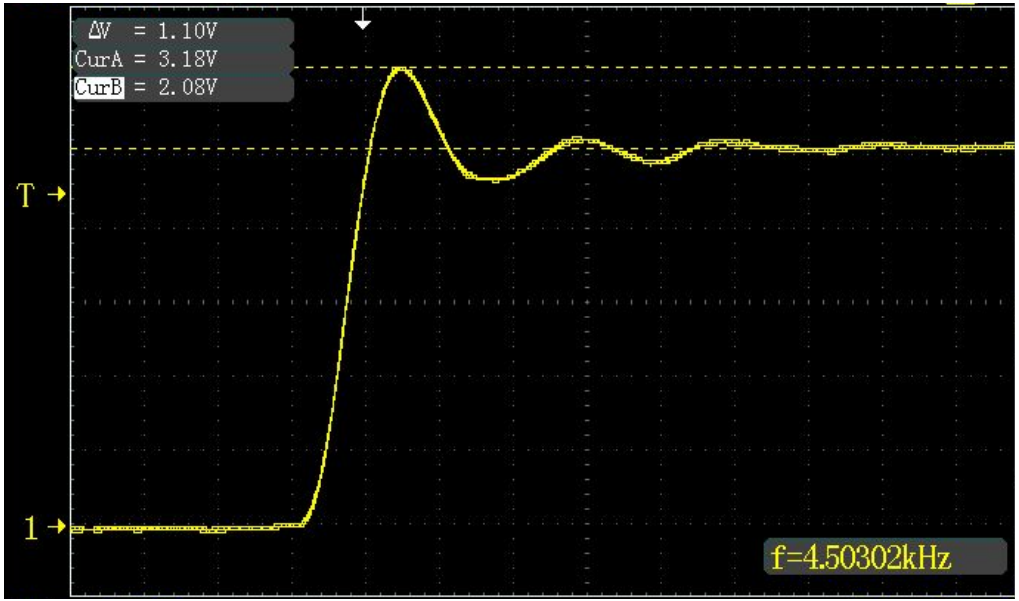


Digital

Credit:
https://commons.wikimedia.org/wiki/File:Analog_clock_animation.gif
https://commons.wikimedia.org/wiki/File:Digital.gif

What is digital?

Analog view:
Voltage



Credit:
https://learn.sparkfun.com/tutorials/how-to-use-an-oscilloscope/using-an-oscilloscope

What is digital?

Analog view:
Voltage

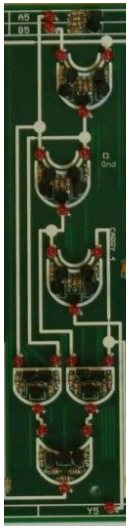
Digital view:
high (1) / low (0)

Credit:
https://learn.sparkfun.com/tutorials/how-to-use-an-oscilloscope/using-an-oscilloscope

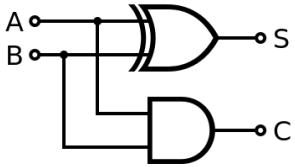
Digital logic in computer architecture

Credit:
https://www.megaprocessor.com/index.html

Digital logic in computer architecture



PCB board



Logic Gate

S = A xor B
C = A and B

Boolean Algebra

Credit:
https://zh.wikipedia.org/zh-hk/File:Half_Adder.svg
https://www.megaprocessor.com/index.html

Boolean Algebra

- Variables: Eithor 0 or 1
- Operators: AND (*), OR (+), XOR (^) ...

S = A xor B
C = A and B

Truth table

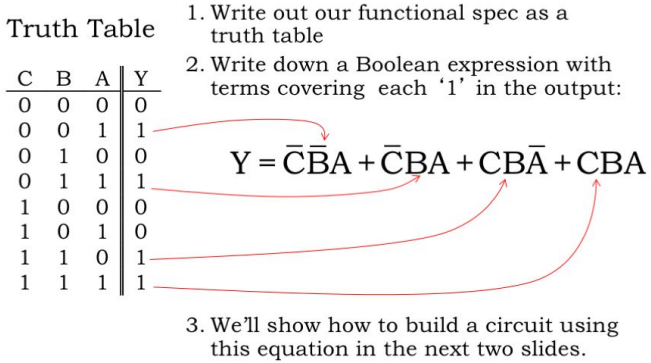
A way to list all possible input and corresponding output

A	B	A + B
0	0	0
0	1	1
1	0	1
1	1	1

A	B	A * B
0	0	0
0	1	0
1	0	0
1	1	1

Karnaugh-map

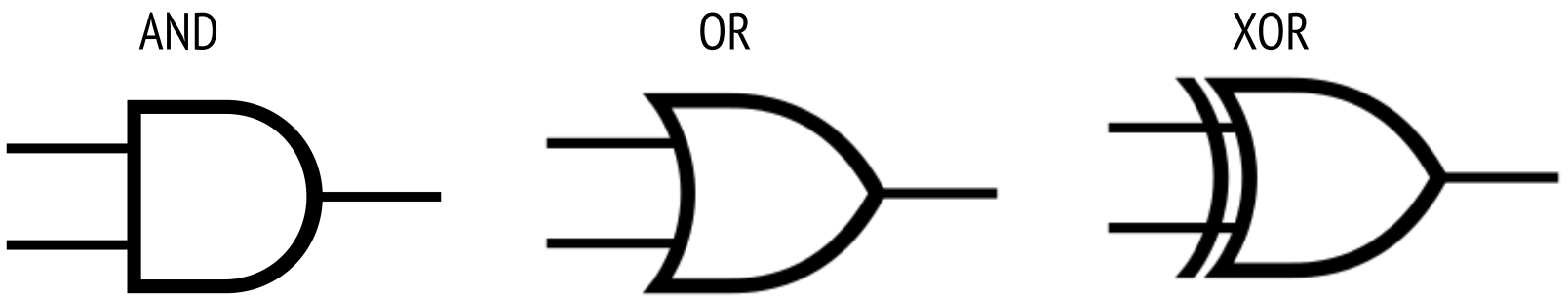
- A way to turn truth table into simplified boolean algebra expressions
- For more details: https://www.youtube.com/watch?v=RO5alU6PpSU



Using K-map: Y = AC' + BC

Logic gates

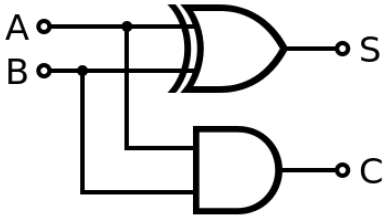
- Physical implementation of operations of boolean algebra
- Logic gate examples:
  - Semiconductor: https://www.youtube.com/watch?v=sTu3LwpF6XI
  - Water: https://www.youtube.com/watch?v=IxXaizglscw&t=157s



Combinational and Sequential Logic

- Combinational
  - output only depends on input
- Sequentail
  - output depends on input and memory state
  - seems as combinational logic + register

Combinational Logic

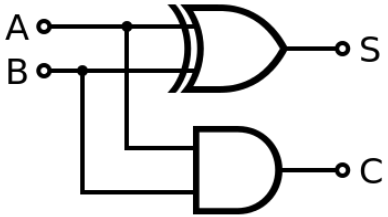


1-bit adder example:
S = A xor B
C = A and B

A	B	S
0	0	0
0	1	1
1	0	1
1	1	0

A	B	C
0	0	0
0	1	0
1	0	0
1	1	1

Combinational Logic

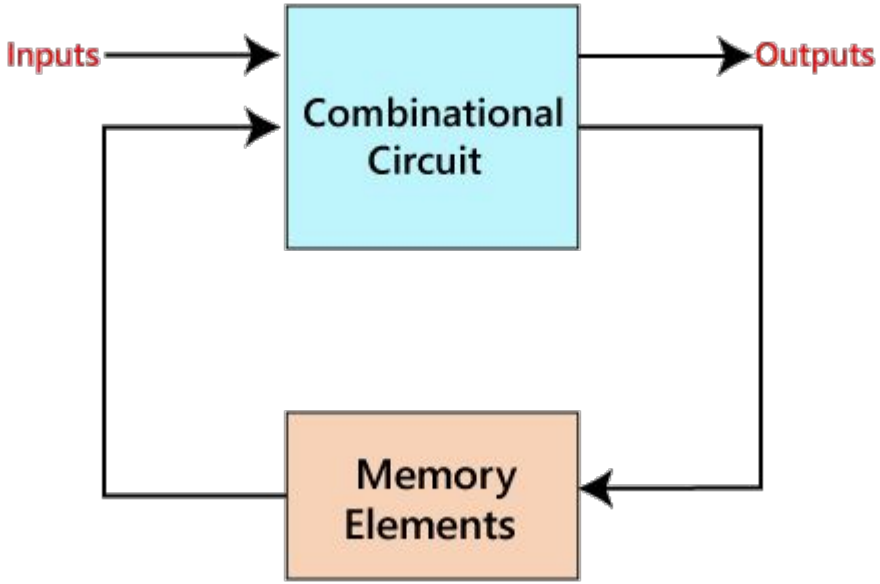


Seem as functions
S = f(A, B)
C = g(A, B)

A	B	S
0	0	0
0	1	1
1	0	1
1	1	0

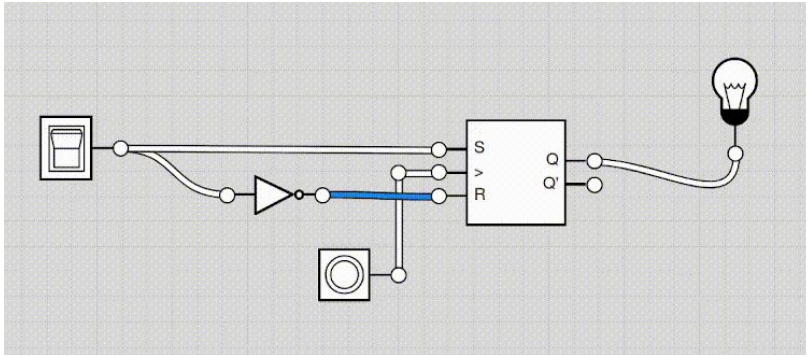
A	B	C
0	0	0
0	1	0
1	0	0
1	1	1

Sequential Logic



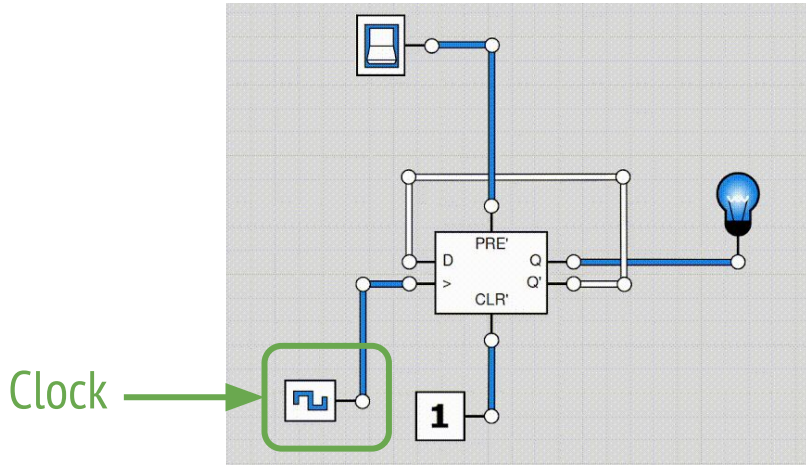
Register

- Components that saves state
- Triggered by another signal (clock mostly)

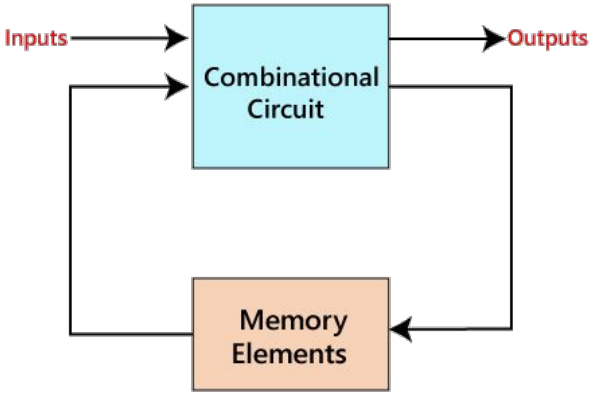


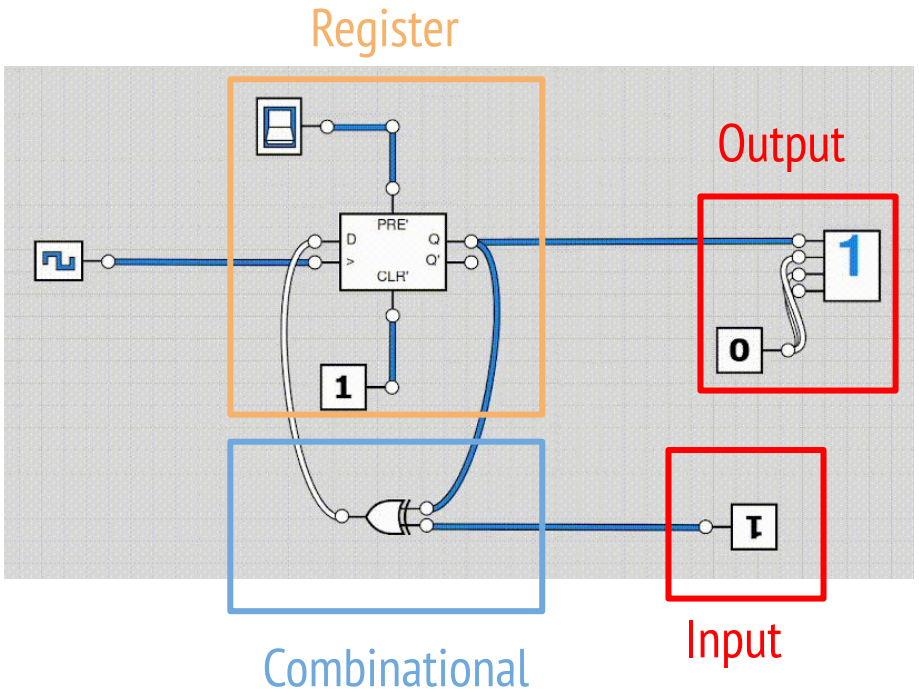
Register

- Components that saves state
- Triggered by another signal (clock mostly)



Sequential Logic

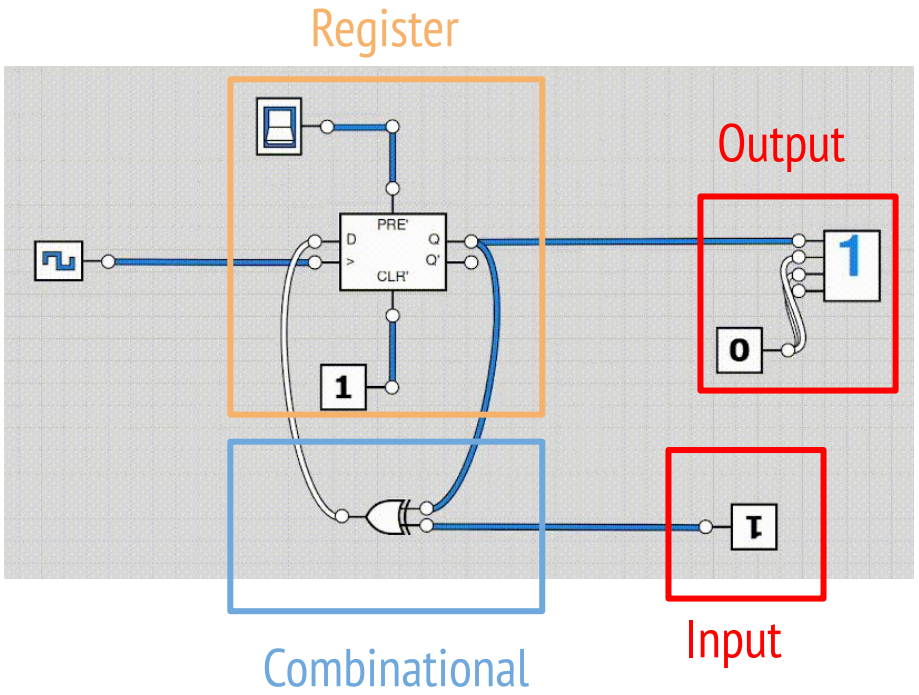




Wave

- X-axis: time
- Y-axis: logic (0 or 1)

Use clock edge to trigger.
(posedge 0 to 1, negedge 1 to 0)



Verilog

- Introduction
- Combinational logic
- Sequential logic
- Testbench

Combinational logic

Let's build a 1-bit adder

- Operators
- Wire
- Module
- Data

Operators

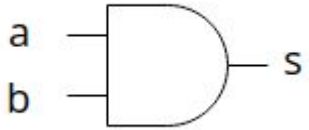
- Basically a logic gate
- More details: https://class.ece.uw.edu/cadta/verilog/operators.html

&	bit-wise AND
^ ^~ or ~^	bit-wise XOR bit-wise XNOR
|	bit-wise OR
&&	logical AND
||	logical OR

Assign

- Wire signals together
- Apply logic operations

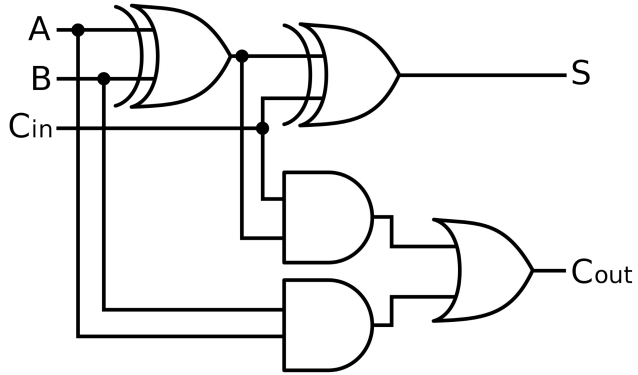
```
assign s = a & b;
```



Input / Output / Wire

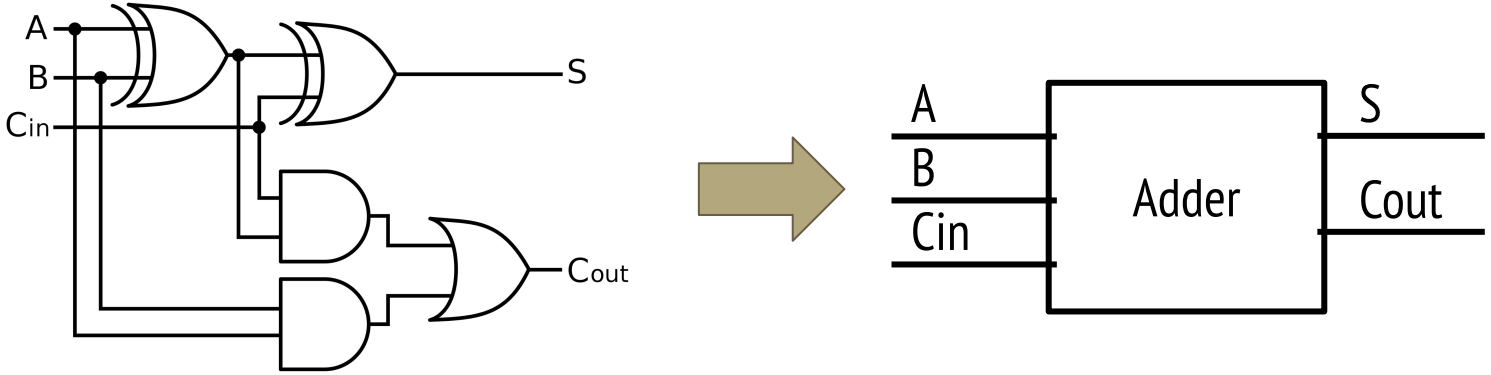
```
 3  input a, b;
 4  input cin;
 5
 6  output s;
 7  output cout;
 8
 9  wire half_sum, carry1, carry2;
10
11  assign half_sum = a ^ b;
12  assign carry1 = a & b;
13  assign carry2 = cin & half_sum;
14  assign s = half_sum ^ cin;
15  assign cout = carry1 | carry2;
```

Refer to sample code "adder_1bit.v"



Credit:
https://zh.m.wikipedia.org/zh-cn/File:Full-adder.svg

Module

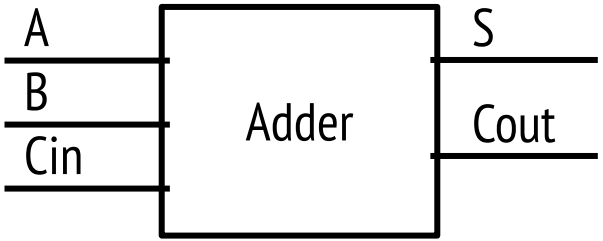


Credit:
https://zh.m.wikipedia.org/zh-cn/File:Full-adder.svg

Module

```
1  module adder_1bit(a, b, cin, s, cout);
2
3  input a, b;
4  input cin;
5
6  output s;
7  output cout;
8
9  wire half_sum, carry1, carry2;
10
11 assign half_sum = a ^ b;
12 assign carry1 = a & b;
13 assign carry2 = cin & half_sum;
14 assign s = half_sum ^ cin;
15 assign cout = carry1 | carry2;
16
17 endmodule
```

Refer to sample code "adder_1bit.v"



Data

- <Width>’<Base><Value>
- Width: width of bits
- Base: b for binary, d for decimal, h for hex
- e.g. 4’b1111 = 4’d15 = 4’hF

Combine 4 1-bit adders into a 4-bit adder

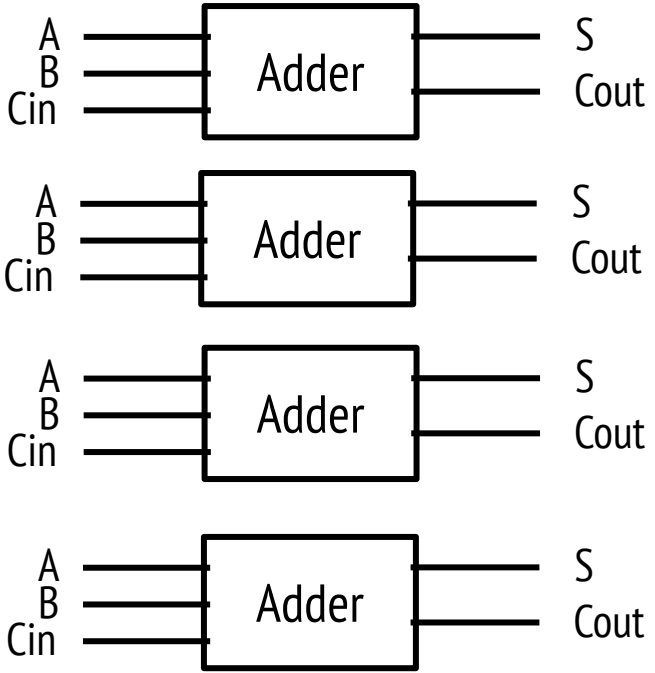
- Module instantiation
- Vector

Module instantiation

- Instantiation predefined module
  - Not like function call, instantiation generates a real hardware component

```
adder_1bit bit0(a[0], b[0], cin, s[0], c0);
adder_1bit bit1(a[1], b[1], c0, s[1], c1);
adder_1bit bit2(a[2], b[2], c1, s[2], c2);
adder_1bit bit3(a[3], b[3], c2, s[3], cout);
```

Refer to sample code "adder_4bit.v"

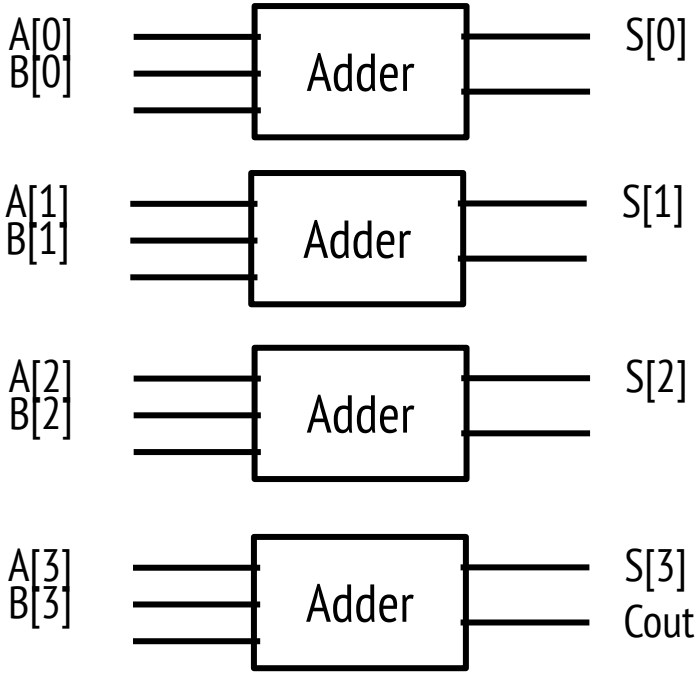


Vector

- Wrap a set of wires into a vector
- Like an array of wires (Be careful of the notation)

```
input [3:0] a, b;
input cin;
output cout;
output [3:0] s;
```

Refer to sample code "adder_4bit.v"

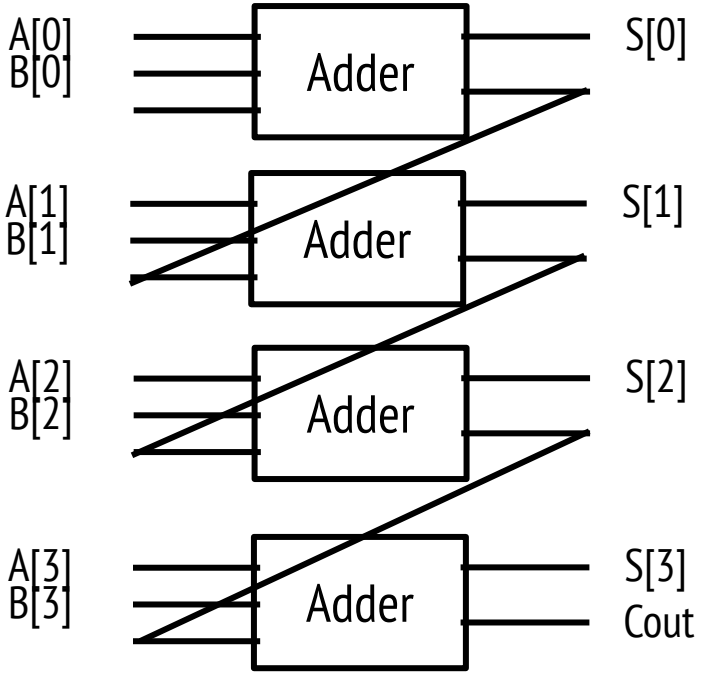


Put them all together

```
module adder_4bit(a, b, cin, s, cout);

input [3:0] a, b;
input cin;
output cout;
output [3:0] s;

wire c0, c1, c2;

adder_1bit bit0(a[0], b[0], cin, s[0], c0);
adder_1bit bit1(a[1], b[1], c0, s[1], c1);
adder_1bit bit2(a[2], b[2], c1, s[2], c2);
adder_1bit bit3(a[3], b[3], c2, s[3], cout);

endmodule
```

Refer to sample code "adder_4bit.v"

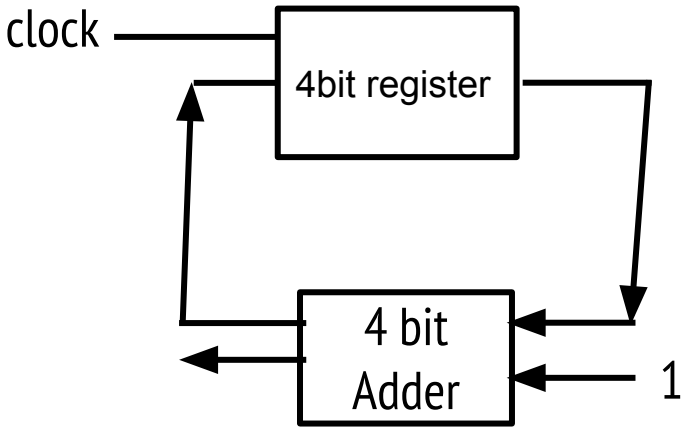
naive ripple carry (slow)



Sequential logic

Next, let's build an accumulator

- 4 bit adder as combinational part
- 4 bit register to save the state

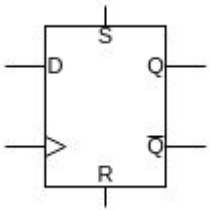


Register

- Use reg to declare a register

```
reg [3:0] register;
```

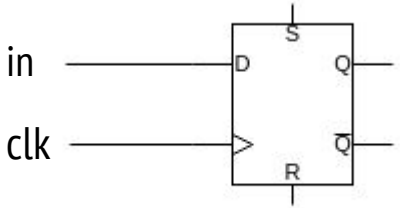
Refer to sample code "register_4bit.v"



Clock

- Use an always block to sense the clock signal that the registers is using

```
 6  reg [3:0] register;
 7
 8  assign out = register;
 9  always @ (posedge clk) begin
10    register = in;
11  end
```

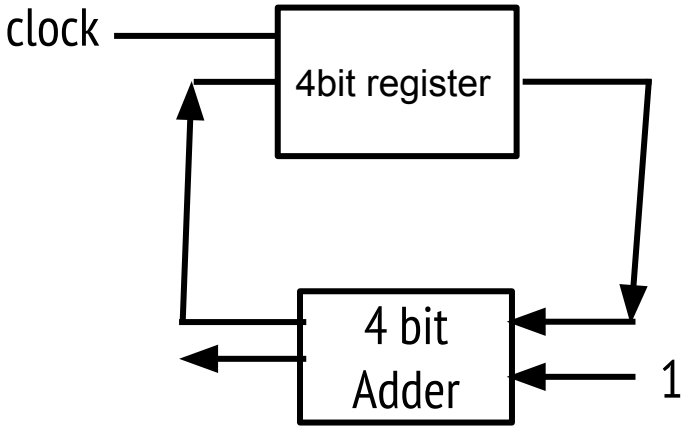


Refer to sample code "register_4bit.v"

Combine the components

```
module accumulator(clk, out);
  input clk;
  output [3:0] out;
  reg [3:0] register;
  wire [4:0] added;

  adder_4bit adder(register, 4'b1, 1'b0, added[3:0], added[4]);

  always @(posedge clk) begin
    out = added[3:0];
  end
endmodule
```

Refer to sample code ".accumulator.v"

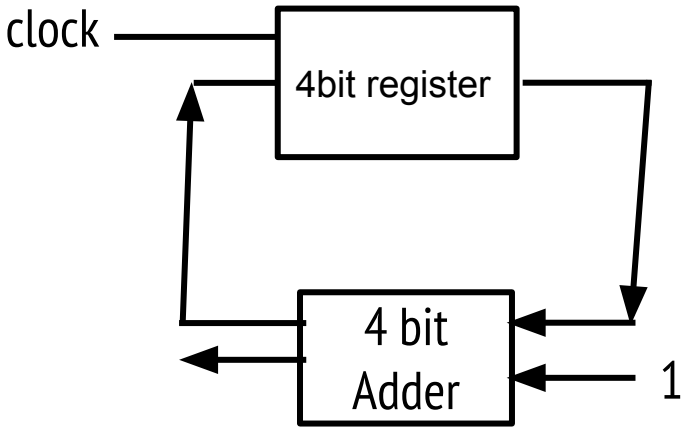


Combine the components

The code doesn't work. Why?

```
1  module accumulator(clk, out);
2    input clk;
3    output [3:0] out;
4    reg [3:0] register;
5    wire [4:0] added;
6
7    adder_4bit adder(register, 4'b1, 1'b0, added[3:0], added[4]);
8
9    always @(posedge clk) begin
10     out = added[3:0];
11   end
12 endmodule
```

Refer to sample code ".accumulator.v"



What is the state in the beginning?

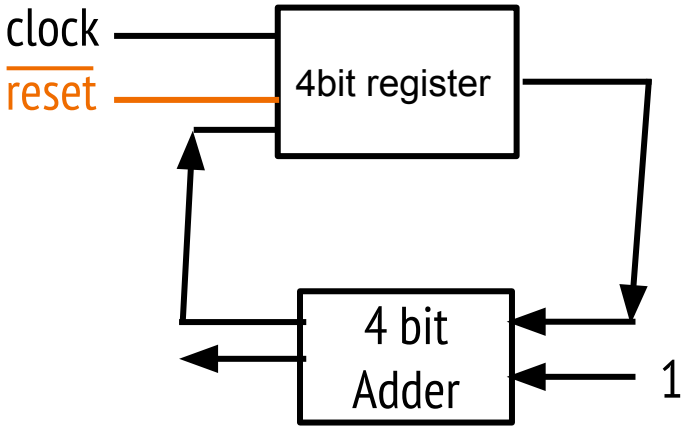
```
module accumulator(clk, out);
  input clk;
  output [3:0] out;
  reg [3:0] register;
  wire [4:0] added;

  adder_4bit adder(register, 4'b1, 1'b0, added[3:0], added[4]);

  always @(posedge clk) begin
    out = added[3:0];
  end
endmodule
```

Refer to sample code ".accumulator.v"

Initial Problem: reg = 4b'xxxx

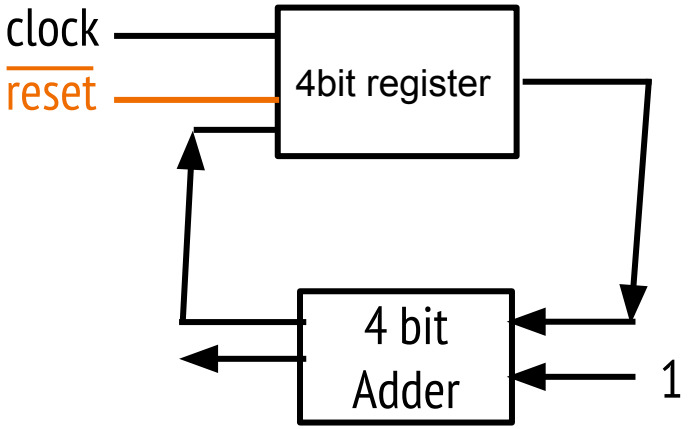
Add a reset signal for initializing

- Reset all register to 0 if reset in 0
- Do what accumulator do otherwise



Add a reset signal for initializing

- Reset all register to 0 if reset in 0
- Do what accumulator do otherwise
- To achieve this, we need a multiplexer

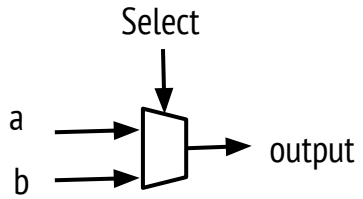


Multiplexer

- Use if statement in always block

```
 8  always @ (*) begin
 9  ··if (sel)
10  ····out = a;
11  ··else
12  ····out = b;
13  end
```

Refer to sample code "multiplexer.v"

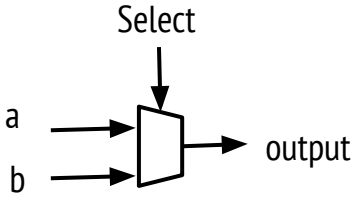


Multiplexer

- Use ? :

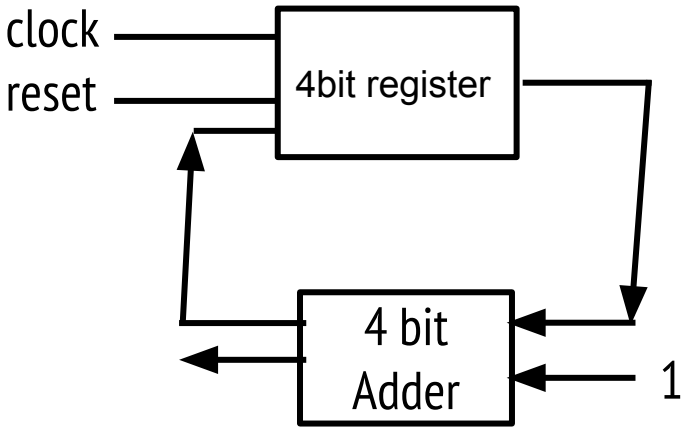
```
3  input a, b, sel;
4  output out;
5
6  assign out = sel ? a : b;
```

Refer to sample code "multiplexer.v"



Add a reset signal for initializing

```
module accumulator(rst, clk, out);
  input rst;
  input clk;
  output [3:0] out;
  reg [3:0] register;
  wire [4:0] added;

  assign out = register;
  adder_4bit adder(register, 4'b0001, 1'b0, added[3:0], added[4]);

  always @ (posedge clk or negedge rst) begin
    if (rst)
      register = added[3:0];
    else
      register = 4'b0;
  end
endmodule
```



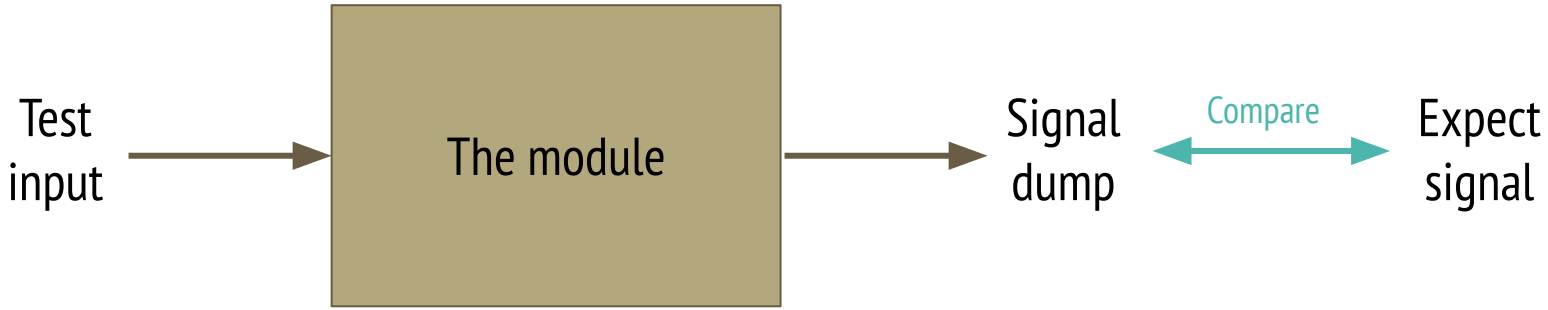
Refer to sample code "accumulator.v"

Testbench

Testbench

- To test your verilog module
- There is something that is non-synthesizable
  - e.g. timing, variable dump...

How can we test a module?



Testbench workflow

1. Initialize the testing environment
2. Send the test inputs
3. Finish the test

Testbench workflow

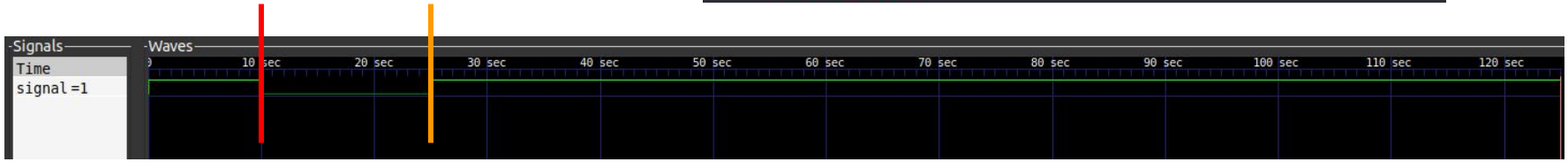
1. Initialize the testing environment
2. Send the test inputs
3. Finish the test

```
module sample_test;

reg signal;

initial begin
  //dump signal into "sample_test.vcd"
  $dumpfile("sample_test.vcd");
  //dump all signals
  $dumpvars;
  signal = 1'b1;
  #10
  signal = 1'b0;
  #15
  signal = 1'b1;
  #100
  $finish;
end

endmodule
```

Refer to sample code "sample_test.v"

Testbench workflow

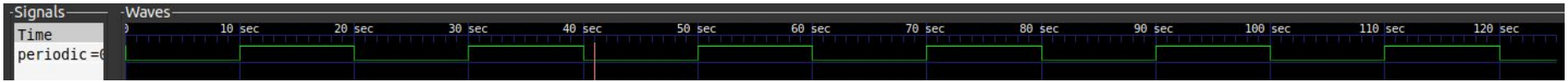
```
1  module sample_test;
2
3  reg signal;
4
5  initial begin
6    // dump signal into "sample_test.vcd"
7    $dumpfile("sample_test.vcd");
8    // dump all signals
9    $dumpvars;
10   signal = 1'b1;
11   #10
12   signal = 1'b0;
13   #15
14   signal = 1'b1;
15   #100
16   $finish;
17 end
18
19 endmodule
```



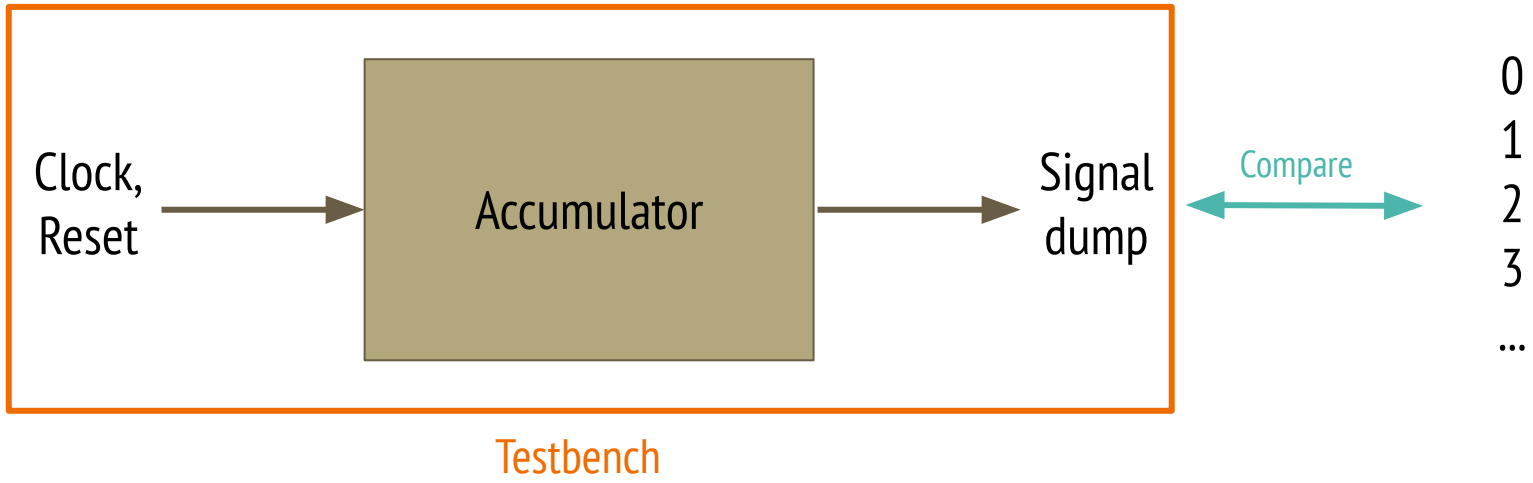
Testbench workflow

- Use always block for periodic signal

```
1  module sample_test;
2
3  reg signal;
4  reg periodic;
5
6  initial begin
7    // dump signal into "sample_test.vcd"
8    $dumpfile("sample_test.vcd");
9    // dump all signals
10   $dumpvars;
11   signal = 1'b1;
12   periodic = 1'b0;
13   #10
14   signal = 1'b0;
15   #15
16   signal = 1'b1;
17   #100
18   $finish;
19 end
20
21 always #10 periodic = ~periodic;
22
23 endmodule
```



Testbench of accumulator



Testbench workflow

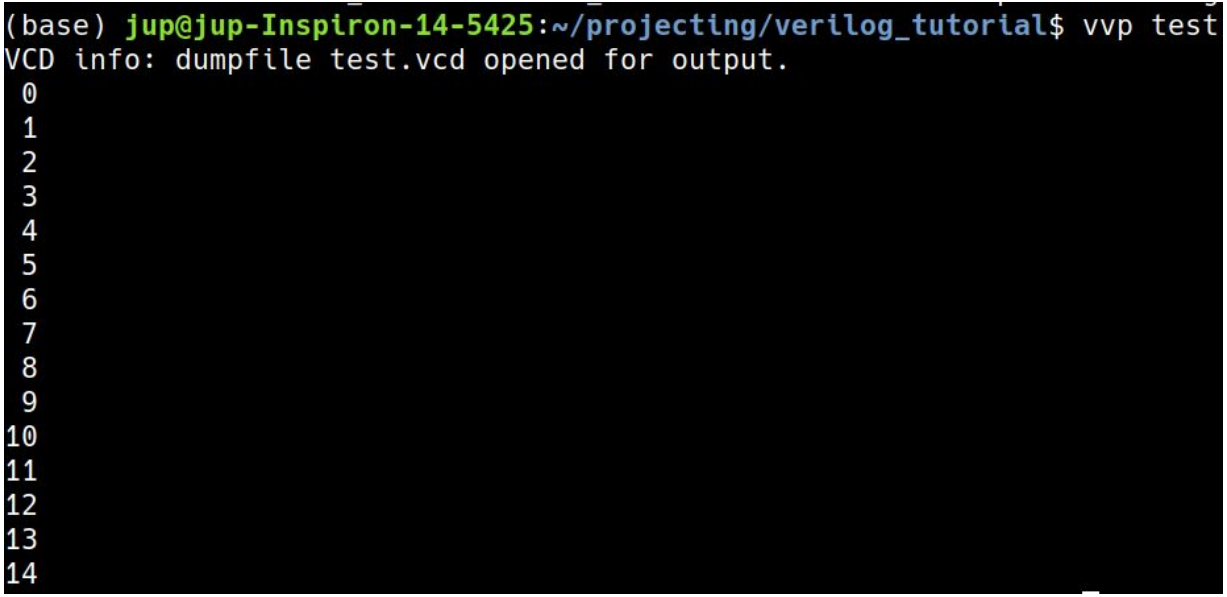
1. Reset accumulator by reset signal
2. Generate an 1hz clock signal
3. Display the value of the accumulator every cycle
4. Finish the test after 30 sec

Testbench workflow

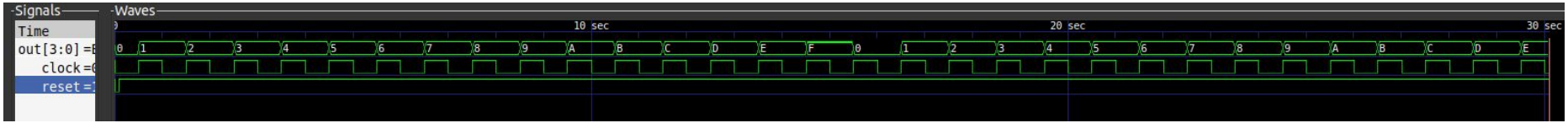
1. Reset accumulator by reset signal
2. Generate an 1hz clock signal
3. Display the value of the accumulator every cycle
4. Finish the test after 30 sec

```
module test;

reg reset, clock;
wire [3:0] out;

accumulator acc(reset, clock, out);

initial begin
  $dumpfile("test.vcd");
  $dumpvars;

  clock = 1'b0;
  reset = 1'b0;

  #0.1
  reset = 1'b1;

  #30
  $finish;
end

always #0.5 begin
  clock = ~clock;
  if (clock)
    $display("%d", out);
end

endmodule
```

Testbench output



Wave


Signals
Waves
Time
out[3:0]
clock
reset
10 sec
20 sec
30 sec
0 1 2 3 4 5 6 7 8 9 A B C D E F 0 1 2 3 4 5 6 7 8 9 A B C D E

Tool usage

iveriog

- iverilog [-o output_name] source-file(s)
- e.g. iverilog -o test source1.v source2.v

vvp

- vvp file_name

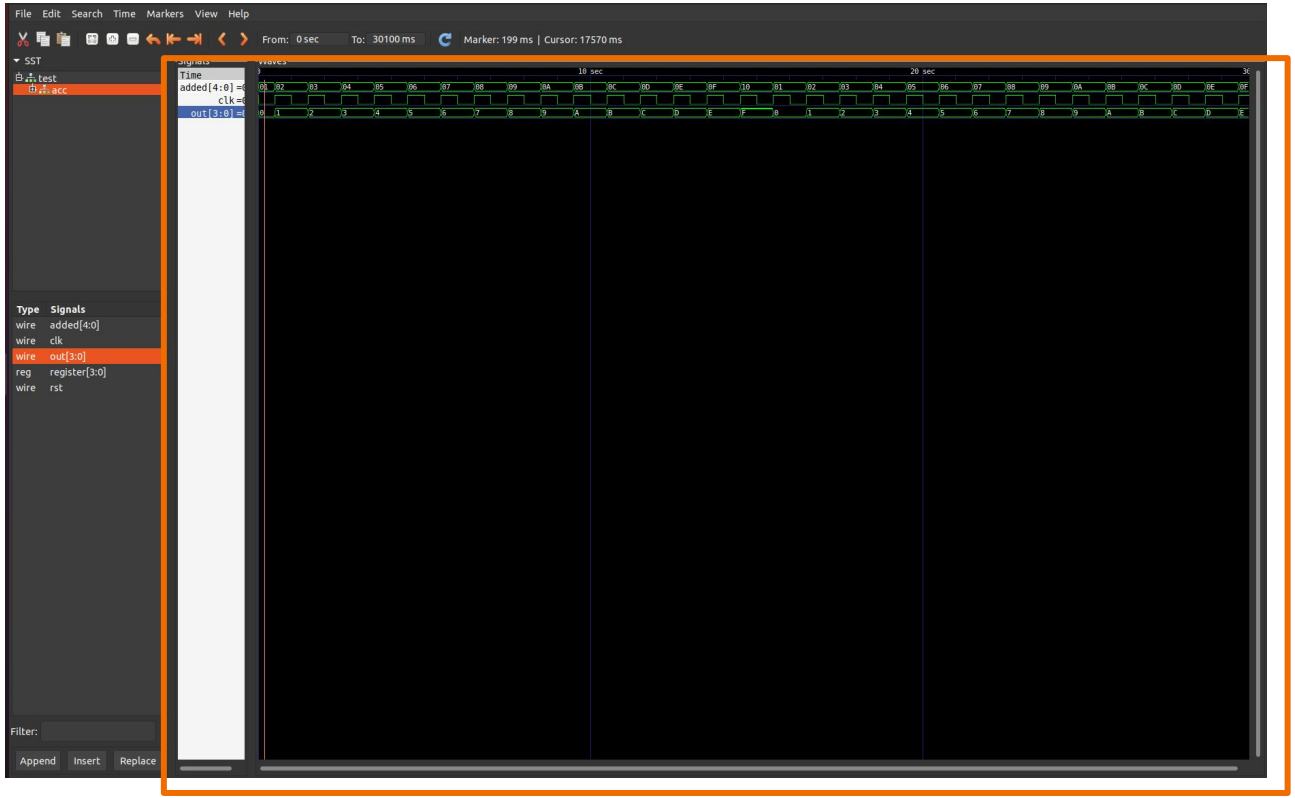
GTKwave

- gtkwave filename

GTKwave

Modules

Signals of the module



wave